VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

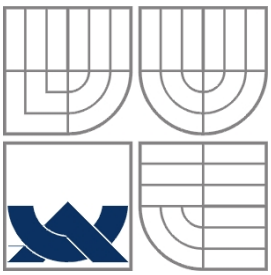
SIMULÁTOR BDI AGENTŮ A OKOLNÍHO PROSTŘEDÍ S PŘEKÁŽKAMI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

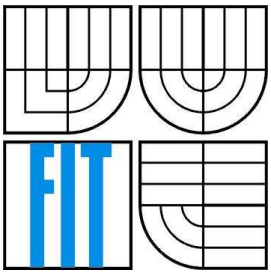
AUTOR PRÁCE
AUTHOR

Bc. Petr Matějíček

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEM

SIMULÁTOR BDI AGENTŮ A OKOLNÍHO PROSTŘEDÍ S PŘEKÁŽKAMI

SIMULATOR OF BDI AGENTS AND SURROUNDING ENVIRONMENT WITH OBSTACLES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Petr Matějíček

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Jan Horáček

BRNO 2011

Abstrakt

Tato práce popisuje implementaci simulátoru multiagentního systému. Vysvětluje základní pojmy agentních a multiagentních systémů. Implementace simulátoru je řešena jako samostatný objekt, umožňující připojovat různé modely prostředí a agenty, kteří v tomto prostředí provádejí akce. V poslední části práce je provedeno vyhodnocení několika různých typů spolupráce agentů.

Abstract

This term project desribes simulator of multiagent system implementation. It explains basic concepts of agent and multiagent systems. Simulator is implemented as a separated object, which allows to join various models of environment and agents acting in this environment. Evaluation of several types of agent cooperation behavior is in last section of this project.

Klíčová slova

Agent, BDI agent, simulátor, simulace, prostředí, agentní systém, multiagentní systém, báze znalostí.

Keywords

Agent, BDI agent, simulator, simulation, environment, agent system, multiagent system, belief base.

Citace

Petr Matějíček: Simulátor BDI agentů a okolního prostředí s překážkami, diplomová práce, Brno, FIT VUT v Brně, 2011

Simulátor BDI agentů a okolního prostředí s překážkami

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením Ing. Jana Horáčka
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Petr Matějíček
24.5.2011

Obsah

1 Úvod

Cílem této práce je vytvořit simulátor multiagentního systému v jazyce C++. Důraz je kladen na možnost snadné změny celého prostředí, případně agentů v něm obsažených. Simulátor je tvořen třemi částmi: modelem prostředí, modelem chování agentů v tomto prostředí a řídícím rozhraním.

Model prostředí je navržen jako 2D mřížka s překážkami. Agenti v tomto prostředí jsou rozděleni do dvou týmů, které spolu soupeří. Úloha, ve které soupeří, je stanovena tak, aby agenti mohli spolu spolupracovát a zvájemně si pomáhat pro dosažení jejich cíle. Vizualizace prostředí je provedena pomocí grafické knihovny OpenGL. Více o modelu prostředí lze nalézt v kapitole **3**.

Agenti jsou implementováni jako tzv. BDI agenti (Beliefs Desires Intentions). Takoví agenti vnímají prostředí okolo sebe a na základě těchto vjemů vytváří plány činností. Výsledkem provádění plánů je akce, kterou daný agent v tomto prostředí provede. Popis BDI agentů lze nalézt v kapitole **2**.

Řídící rozhraní představuje abstraktní simulátor, který řídí činnost celého systému. Spojuje vytvořené agenty a model prostředí, a poskytuje základní operace pro přístup vytvořeného modelu k informacím obsažených v agentech. Implementace simulátoru je popsána v kapitole **4,** implementace chování agentů v kapitole **5**. Závěrečná kapitola **6** obsahuje vyhodnocení efektivity různých typů agentů pro řešení úlohy v několika testovacích prostředích.

2 Agentní a multiagentní systémy

Tato kapitola popisuje základní pojmy agentních a mutliagentních systémů. Je převzata z [5] a [6].

2.1 Umělý agent

Umělý agent je člověkem vytvořené dílo, které v prostředí do kterého je umístěno jedná samostatně ve prospěch svého klienta.

Nejdůležitější vlastnost agenta, která ho odlišuje od jiných objektů, je schopnost samostatně jednat. Agent musí být schopen v prostředí nejenom řešit zadanou úlohu, ale musí zvládnout vyřešit bez zásahu člověka i různé konflikty, které mohou v průběhu jeho činnosti nastat. Mezi další vlastnosti agenta patří:

- **Reaktivita** – Agent musí umět pohotově reagovat na změny prostředí.
- **Proaktivita** – Agent musí být schopen samostatně generovat akce vedoucí ke splnění jeho cíle.
- **Sociálnost** – V multiagentním prostředí agent musí být schopen sociálního chování, komunikovat s ostatními agenty, společně řešit konflikty, spolupracovat.

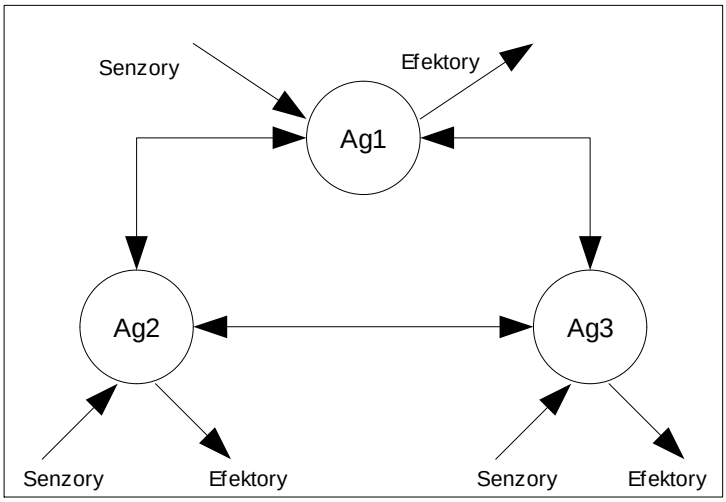
2.2 Prostředí

Agenti jsou umístěni do nějakého prostředí. Prostředí obsahuje vnitřní stavy, které ho reprezentují. Lze ho definovat jako:

- **Spojité a diskrétní** - Prostředí lze dělit na spojité nebo diskrétní, podle definované časové množiny. Protože reálný svět je prostředí spojité a počítačový systém je diskrétní, je nutné při tvorbě agentích systémů na počítači prostředí diskretizovat.
- **Statické a dynamické** - Stav statického prostředí se nemění, pokud agent neprovádí žádnou činnost. Dynamické prostředí může svůj stav změnit i pokud agenti nepracují.
- **Deterministické a nedeterministické** - V deterministickém prostředí je nový stav dán aktuálním stavem a akcí prováděnou agentem. Každá akce agenta má právě jeden efekt.
- **Epizodní a neepizodní** - Chování prostředí lze rozdělit na epizody. Tyto epizody jsou na sobě nezávislé a po ukončení činnosti jedné epizody, lze začít novou od počátečního stavu.
- **Přístupné a nepřístupné** - Je-li prostředí přístupné, agent je kdykoliv schopen získat úplnou informaci o stavu prostředí.
- **Strategické** - Prostředí je strategické, pracuje-li v prostředí více agentů. Přestože prostředí může být statické, činnost ostatních agentů může způsobit, že se prostředí agentovi jeví jako dynamické (agent neprovedl žádnou činnost a stav prostředí se přesto změnil akcí jiného agenta).

2.3 Agentní a multiagentí systém

Agentní systém je definován jako dvojice obsahující prostředí a agenta. Komunikační rozhraní agenta obsahuje **senzory** a **efektory.** Agent svými senzory vnímá stav prostředí. Na základě tohoto stavu vytvoří akci, kterou ovlivňuje stavy tohoto prostředí. Tuto akci provede pomocí svých efektorů. Na základě této akce a aktuálního stavu prostředí je proveden přechod do stavu nového. Multiagentní systém je agentní systém, který obsahuje v prostředí více než jednoho agenta:



Obr 2.1: Multiagentní systém.

Kromě ovlivňování stavu prostředí svými efektory, jsou agenti schopni mezi sebou komunikovat a spolupracovat. Agenti tedy mají možnost si mezi sebou zasílat zprávy.

2.4 BDI agent

BDI agent, jenž je v této práci implementován, je jeden z možných způsobů realizace chování agenta. Zkratka BDI znamená **Beliefs** (představy) **Desires** (přání) **Intentions** (záměry). Řízení agenta probíhá na základě těchto údajů. Představy obsahují agentovy znalosti o prostředí. Přání reprezentují to, co by si agent svou činností přál dosáhnout. Záměry představují agentovu činnost vedoucí k naplnění některého ze svých přání.

Agent ze svých přání stanoví cíl své činnosti a vytvoří si záměry vedoucí k dosáhnutí tohoto cíle. Pro své záměry vytvoří plán činnosti vedoucí k jejich dosažení.

2.4.1 Abstraktní kontrolní smyčka

Abstraktní kontrolní smyčka představuje jednu z možností jak řídit činnost BDI agentů. Pseudoalgortimus této smyčky vypadá následovně[6]:

```
B = B0;
I = I0;
p = null;
while 1 do
      p = see();
      B = update_beliefs(B,p);
      if(reconsider(B,I)) then
            D = options(B,I);
            I = filter(B,D,I);
            if (not sound(π)) then
                  π = plan(B,I);
            end-if
      end-if
      if (not empty(π)) then
            a = hd(π);
            execute(a);
            p = tail(π);
      end-if
end-while
```

Na začátku agent obsahuje počáteční představy o okolí a nějaký záměr, kterého se snaží dosáhnout. V řídící smyčce je implementována činnost agenta:

1. Agent pozoruje svými senzory okolí a na základě tohoto pozorování může upravit svoje představy (`update_beliefs`).
2. Pokud současné představy zmemožňují splnění zvoleného záměru, nebo jsou k dispozici nové možnosti splnění agentova cíle, agent se může rozhodnout přehodnotit svoje aktuální záměry (`reconsider`)
3. Agent si na základě svých představ a aktuálních záměrů vytvoří nové cíle (`options`)
4. Z těchto cílů agent vytvoří nové záměry (`filter`)
5. Pokud současný plán agenta není možné aplikovat k dosáhnutí nových záměrů (`not sound`), agent vytvoří plán nový (`plan`)
6. Agent provede první akci plánu (`execute`) a zbytek plánu uloží jako současný plán pro další krok smyčky

2.4.2 Příklad realizace BDI agentů

Jedním z programovacích jazyků využívajících BDI architekturu je JASON (http://jason.sourceforge.net/Jason/Jason.html). Je napsán v jazyce JAVA a je založen na logickém programování (podobně jako například jazyk PROLOG).

JASON reprezentuje představy (beliefs) jako tzv. bázi znalostí. Tato báze slouží jako databáze pro uložení různých informací (představách agenta). Agent se může dotazovat své báze na její obsah a podle něj vybírat ze své množiny plánů plán, který bude provádět. Plán je v jazyce napsán jako posloupnost akcí agenta. Mezi tyto akce může patřit úprava a dotazy na bázi znalostí (například pro agentovy vlastní záznamy), volání dalších plánů (například pokud má plán nějaké další podplány), nebo přímo akce, kterými ovlivňuje prostředí. Nemožnost splnění jakékoli akce plánu znamená neúspěšnost splnění celého plánu a agent musí přejít na plán jiný.

Implementace agentů v této práci částečně vychází z interpretace BDI agentů v jazyce JASON. Na rozdíl od logického programování agentů v JASONu, implementace plánování a činnosti agenta je řešena imperativním způsobem. Báze představ je založena na podobném principu jako v JASONu (unifikace představ v bázi spolu s dotazy na bázi). Popis návrhu agenta lze nalézt v kapitole **3.2** a popis implementace v kapitole **4.1**.

2.4.3 Příklad multiagentního systému realizovaného v jazyce JASON

V této kapitole je ukázán příklad jednoduchého multiagentnáho systému, vytvořeného v zmiňovaném jazyce JASON. Uvažujme dva agenty se jmény **customer** a **shop**:

- **customer**

```
money(20).

!buy.
+!buy<-
       .send("shop",achieve,get_price("book")).
+!price(X): money(Y) & Y >= X <-
       .send("shop",achieve,buy("book")); -money(Y);+money(Y-X).
+!price(X).
+!take(X)<-
       +bought(X).
```

- **shop**

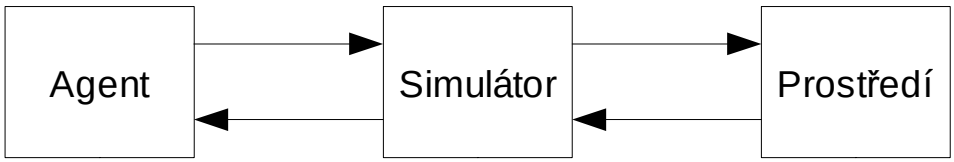
```
stock("book",10).

+!get_price(X): stock(X,Y) <-
       .send("customer",achieve,price(Y)).
+!get_price(X).
+!buy(X)<-
       -stock(X,Y); .send("customer",achieve,take(X)).
```

Tento jednoduchý systém modeluje situaci nakupujícího a prodávajícího agenta. Nakupující agent (customer) má na začátku určitou částku peněz, zadanou hondotou `money`. Záměrem agenta je koupit si knihu. Volá tedy plán `buy` ve kterém pošle agentovi prodávajícímu agentovi zprávu o provedení plánu `get_price`. Prodávající agent (shop) zkontroluje zda je požadované zboží dostupné a pošle zpět zprávu o provedení plánu `price`. Nakupující provede kontrolu zda má dostatek peněz (`Y >= X`), a pokud ano, zašle zprávu o provedení plánu `buy`. Po zaslání této zprávy odečte od hodnoty `money` patřičnou částku ceny knihy (nejdříve záznam odstraní a pak ho nahradí upraveným). Prodávající agent provede odebrání záznamu `stock` a pošle nakupujícímu agentovi plán, kterým je uložena kniha do databáze jako `bought("kniha")`.

Tento příklad ukazuje zmíněnou podobnost jazyka JASON s jazykem prolog, zejména unifikaci proměnných. Více o programování v jazyce JASON lze nálezt v [1] a [2].

3 Návrh simulátoru multiagentního systému

Celý simulátor je navržen jako 3 objekty, které spolu vzájemně komunikují.

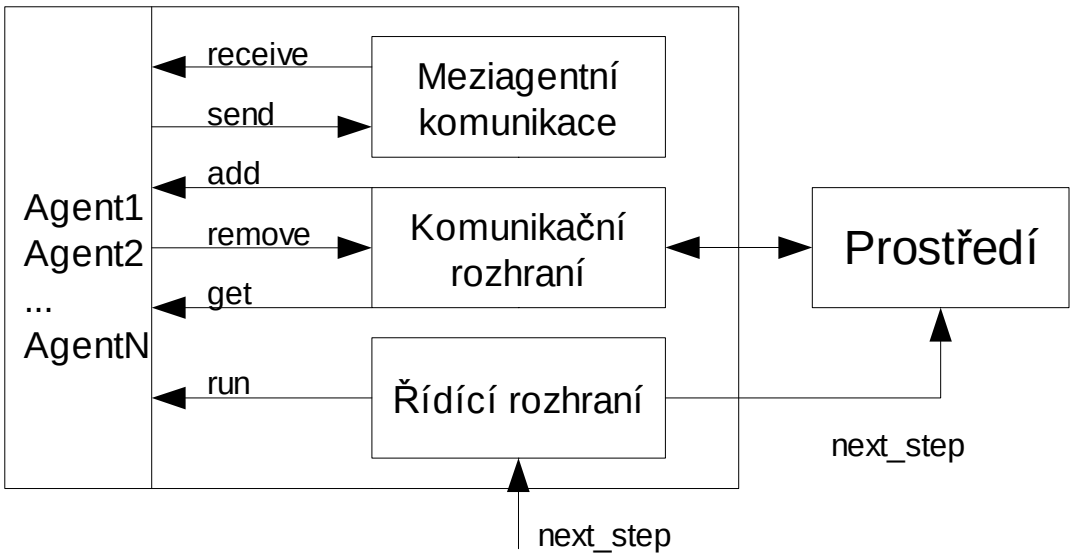


Obr. 3.1: Blokové schéma systému.

Obrázek **3.1** zobrazuje blokové schéma navrhovaného systému. Systém se skládá ze tří částí – **Agent**, **Simulátor** a **Prostředí**. Blok Agent reprezentuje jednotlivé agenty nacházející se v systému. Blok Prostředí reprezentuje model prostředí, ve kterém simulace probíhá. Blok Simulátor je propojovací vrstva mezi agenty a prostředím, a řídí činnost celého programu. Simulátor je implementován tak, aby bylo možné bloky Prostředí a Agent podle potřeby měnit nebo upravovat, a tak snadno měnit řešenou úlohu.

3.1 Blok Simulátor

Tento objekt představuje rozhraní mezi prostředím a agenty, kteří se v tomto prostředí nachází. Prostředí komunikuje s agenty přes komunikační rozhraní, které zapouzdřuje agentovy metody pro přístup do báze znalostí. Simulátor lze znázornit schématem na obrázku **3.2**:



Obr. 3.2: Blokové schéma simulátoru.

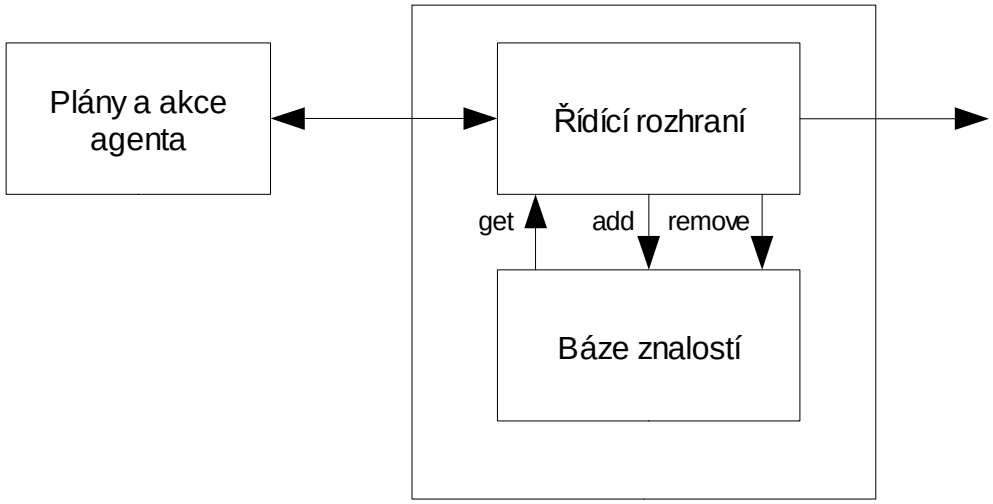
Následující body popisují jednotlivé bloky ve schématu:

- **Agenti** - Ukládá všechny agenty vytvořené v simulaci.
- **Komunikační rozhraní -** Poskytuje prostředí metody, které mohou komunikovat s bází znalostí vytvořených agentů - přidání (add), smazaní (remove) a získaní (get) znalosti z báze znalostí. Tato komunikace zahnuje i získání akce agenta, kterou agent vygeneruje v rámci svého plánování.
- **Meziagentní komunikace -** Meziagentní komunikace je rozhraní, pomocí kterého mohou agenti zasílat zprávy jiným agentům v systému.
- **Řídící rozhraní -** Obsahuje metody, které řídí činnost celého systému – spouští agenty a simulaci prostředí.

Činnost simulátoru je krokována voláním `next_step`. Tato metoda spustí činnost agentů. Po ukončení jejich plánování spustí krok v simulovaném prostředí, jež tyto plány zpracuje. Opakovaným voláním `next_step` dochází k samotné simulaci multiagentního systému.

3.2 Blok Agent

Je objekt reprezentující vytvořeného agenta. Blokové schéma agenta je zobrazeno na obrázku **3.3**:



Obr. 3.3: Blokové schéma agenta.

Hlavní prvek je báze znalostí. Obsahuje veškeré znalosti agenta o prostředí. Je používána agentem pro uložení vlastních znalostí získaných na základě pozorování prostředí. Řídící rozhraní poskytuje metody pro přístup k bázi (získání znalosti, přidání znalosti, smazání znalosti). Kromě práce s bází znalostí, obsahuje tento objekt metodu zjištění jména agenta a další pomocné vnitřní metody**.**

Blok *Plány a akce agenta* obsahuje metody pro řízení činnosti agenta. Čtením informací ze své báze představ zjišťuje informace o okolním prostředí. Výsledkem plánování je akce, kterou agent vykoná.

3.2.1 Báze znalostí

Báze znalostí je navržena jako databáze záznamů ve tvaru:
`name(arg1, arg2, ..., argN)`

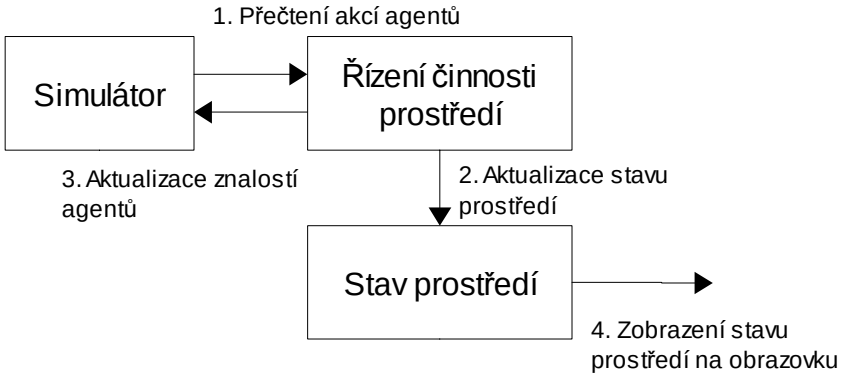
kde `name` je jméno představy a `arg1..argN` jsou parametry znalosti. Vyhledávání záznamů probíhá pomocí unifikací. Například, je-li agentova báze napěna:

```
animal(spider,8)
animal(dog,4)
animal(cat,4)
```

První argument značí jméno a druhý počet nohou zvířete. Při hledání záznamu zvířete, které má čtyři nohy `animal(X,4)`, je vyhledán v databázi první záznam, u kterého se shodují zadané hodnoty. Neznámé hodnoty jsou unifikovány s nalezeným záznam, tedy výslená nalezená znalost v tomto případě je `animal(dog,4).`

3.3 Blok Prostředí

Obsahuje popis simulovaného prostředí a zpracovává logiku řešené úlohy. Prostředí komunikuje s řídícím rozhraním, přes které získává informace o agentech v systému. Na základě těchto informací upravuje svoje vnitřní stavy a aktualizované hodnoty posílá přes řídící rozhraní agentům zpět. Obrázek **3.4** ukazuje postup činnosti modelu prostředí.

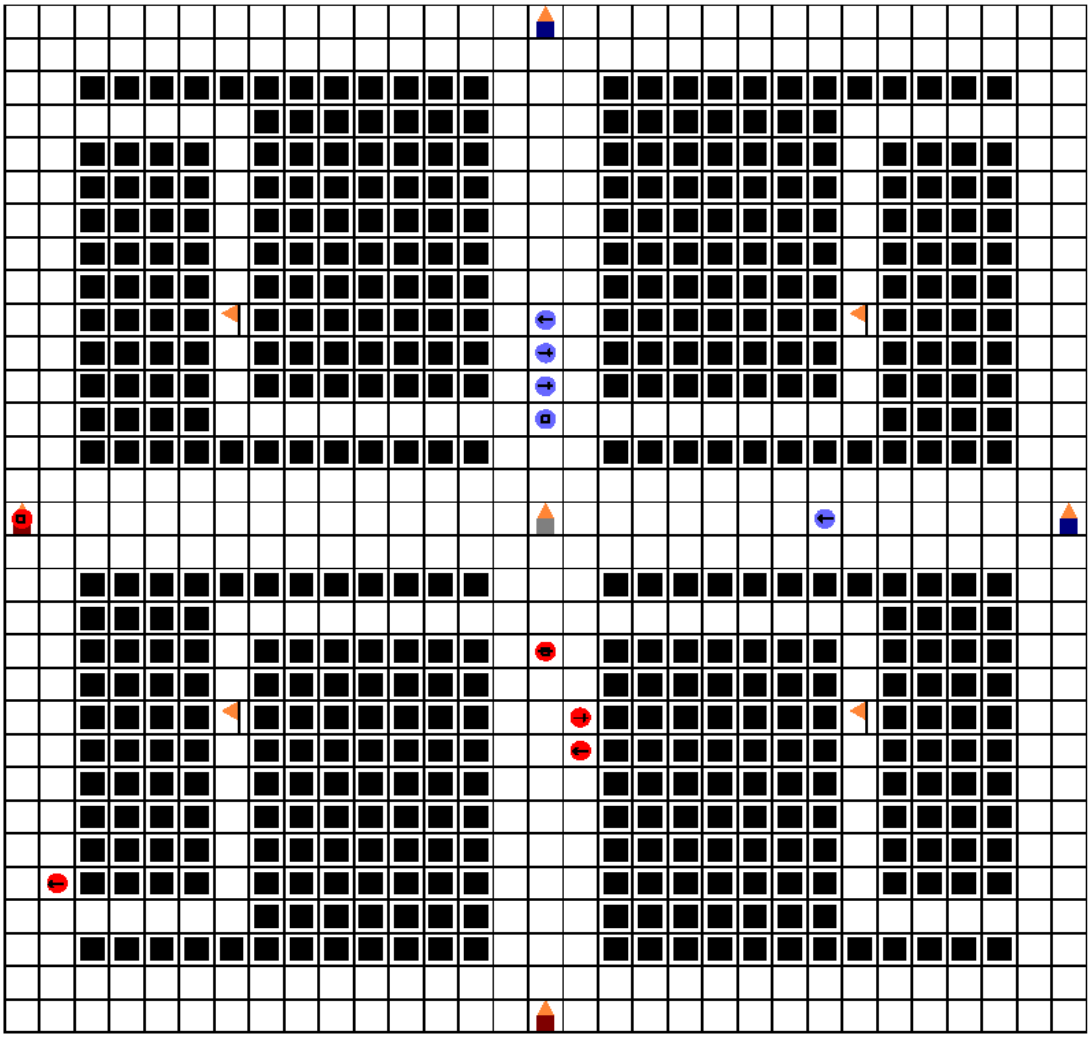


Obr. 3.4: Blokové schéma činnosti modelu prostředí.

3.4 Návrh řešené úlohy a modelu prostředí

Prostředí je definováno jako 2D mřížka o rozměrech 31x31. Agenti se mohou pohybovat po této mřížce v každém kole o jeden krok v horizontálním a vertikálním směru (doleva, doprava, nahoru, dolu). Každé políčko v mřížce může být označeno jako překážka, na které agent nemůže vstoupit.

Zadaná úloha je zvolena tak, aby bylo možné agenty rozdělit do dvou týmů, které proti sobě soupeří. Cílem týmu je dosáhnout určitého počtu bodů dřív než jeho soupeři. Týmy získávají body za obsazovaní strategických pozic na mřížce. Ukázka jak může vypadat prostředí je na obrázku **3.5.**



Obr. 3.5: Ukázka aplikace: Herní mřížka.

Prostředí obsahuje tyto prvky:

Překážky – Jsou políčka, na které se není schopen agent dostat.
Věž – Jsou objekty, které můžou agenti zabírat (přebarvit je na barvu svého týmu).
Vlajka – Je možné ji agentem sebrat a donést na obsazenou věž, vlajka se potom opět obnoví na své původní pozici.
Agenti – Jsou zobrazeni jako červené (první tým) a modré (druhý tým) kolečka.

Každý z definovaných prvků je reprezentován určitými parametry, které ho plně charakterizují:

- **Překážky** – Pozice *x*,*y* na mřížce.
- **Věž** – Pozice *x*,*y* na mřížce, počet životů věže, barva vlastnícího týmu.

- **Vlajka** – Pozice *x*,y na mřížce, počet životů.
- **Agenti** – Pozice *x*,*y* na mřížce, velikost týmu, velikost zranění, počet životů, typ útoku.

Agenti se pohybují v tomto prostředí a snaží se obarvit co nejvíce věží na barvu svého týmu (obsadit věž). Na začátku každého kroku běhu simulátoru je vypočítán přírustek skóre týmu v závislosti na počtu obarvených věží podle vzorce $gain = 5 . 2^x$, kde x je počet obsazených věží. Donesení vlajky na obsazenou věž přidá jednorázový bonus ke skóre podle vzorce $gain = 15 . 2^x$, kde x je opět počet obsazených věží. Po donesení je vlajka vrácena na původní pozici, kde ji může sebrat další agent.

Potkají-li se dva agenti ze soupeřících týmů, může dojít k boji. Pokud je agent v boji poražen, je vrácen zpět na startovní pozici a musí určitou dobu čekat, než bude moci opět začít provádět akce.

V každém kole jsou agentům zasílány do jejich báze znalostí informace o jejich blízkém okolí (4-okolí, včetně samotného místa kde se agent nachází):

- pozice nepřátelských agentů,
- existence vlajky,
- pozice překážek,

Následující znalosti jsou zasílany v každém kole bez ohledu na jejich pozici:

- pozice věží a jejich vlastnictví,
- aktuální skóre hry,
- aktuální velikost týmu agenta,
- pozice agentů z agentova týmu,

Poslední skupina znalostí je zasílana před začátkem simulace a v průběhu se nemění:

- rozměr herní mřížky,
- počet věží,
- počet vlajek,
- pozice vlajek,
- počet agentů v týmu,
- jména agentů v týmu,

Celá simulace končí, dosáhne-li některý z týmů stanovený počet bodů.

3.5 Popis agentů

Jeden agent je v prostředí reprezentován jako skupina o velikosti 5. Velikost skupiny udává sílu agenta pro jeho akce. Např. čím větší skupina, tím rychleji jsou zabírány věže a sbírány vlajky. Agenti jsou popsáni těmito parametry:

- **Velikost skupiny** – Síla agenta.
- **Počet životů** – Počet životů agenta.
- **Velikost zranění** – Počet životů, které je schopen agent v boji ubrat nepřátelskému agentovi.
- **Typ útoku** – Agent může mít 2 různé typy útoku: útok na blízko (agent může útočit pouze na nepřátele nacházející se na stejné pozici) a útok na dálku (agent může zaútočit i na sousední agenty).

Akce, které mohou agenti provádět jsou následující:

- **Pohyb** – Přesun agenta nahoru, dolu, doleva, doprava, není-li na cílovém políčku nějaká překážka.
- **Sebrání vlajky** – Pokud se agent nachází na stejné pozici jako vlajka, může ji sebrat.
- **Položení vlajky** – Pokud se agent nachází na políčku, kde jeho tým vlastní věž, může zde vlajku položit.
- **Zabírání věže** – Pokud se agent nachází na stejném políčku s věží, může se ji pokusit zabrat pro svůj tým.
- **Útok** – Útok na nepřátelského agenta.

Prostředí definuje 3 druhy agentů:

- **Útočný agent**
  - velikost skupiny 5
  - počet životů 1
  - velikost zranění 3
  - útok na blízko

- **Obranný agent**
  - velikost skupiny 5
  - počet životů 3
  - velikost zranění 1
  - útok na blízko

- **Agent útočící na dálku**
  - velikost skupiny 5
  - počet životů 1
  - velikost zranění 2
  - útok na dálku

Agenti se tedy liší svými parametry, které korespondují s jejich typy. Útoční agenti mají největší velikost zranění, naopak obranní agenti mají nejvíce životů. Agenti útočící na dálku mají, na rozdíl od ostatních, hodnoty parametrů nízké, což ale kompenzují schopností útočit na dálku.

3.5.1 Sebrání vlajky

Při pokusu o sebrání vlajky se síla agentovy skupiny odečte od počtu životů této vlajky. Agent vlajku sebere, pokud počet životů vlajky po pokusu o sebrání je <= 0. Např.: Má-li vlajka 10 životů a síla agenta je 5:

1.kolo → 10 – 5 = 5
2.kolo → 5 – 5 = 0 → vlajka je sebrána

3.5.2 Položení vlajky

Akce položení vlajky přičte týmu počet bodů stanovený podle vzorce uvedého v kapitole **3.4**. Vlajka je automaticky vrácena zpět na svou výchozí pozici a její počet životů je nastaven na výchozí (maximální) hodnotu.

3.5.3 Zabírání věží

Při zabírání věží platí to samé, jako při braní vlajek. Každá věž má určitý počet životů. Síla agenta se od něj odečítá. Dosáhne-li počet na hodnotu <= 0, pak je věž přebarvena na barvu týmu. Po zabrání věže je její počet životů resetován na výchozí (maximální) hodnotu.

3.5.4 Útok

Při útocích na jiné agenty určuje síla agenta šanci, se kterou je agent schopen provést úspěšný útok. Úspěšný útok ubere soupeřícímu agentovi takový počet životů, kolik je velikost zranění útočícího agenta. Klesne-li počet životů agenta na 0 nebo méně, je mu snížena síla skupiny o 1 a životy jsou nastaveny zpět na výchozí hodnotu (v případě, že počet životů po zranění byl 0), případně na hodnotu menší (pokud byly jeho životy < 0, odečte se rozdíl). Pokud síla skupiny agenta klesne na 0, je agent zabit. Mrtvý agent je vrácen zpět na startovní pozici týmu a musí čekat zadaný počet kol, než bude moci provádět další akce.

3.5.5 Příklad výpočtu útoku

Vzorec pro výpočet šance na zásah je *chance = 0,2.x*, kde *x* je síla týmu agenta. Maximální síla agenta je 5, tedy při plné síle je šance na zásah 100%. Každé snížení síly o 1 znamená zmenšení šance na zásah o 20 %.
Příklad:

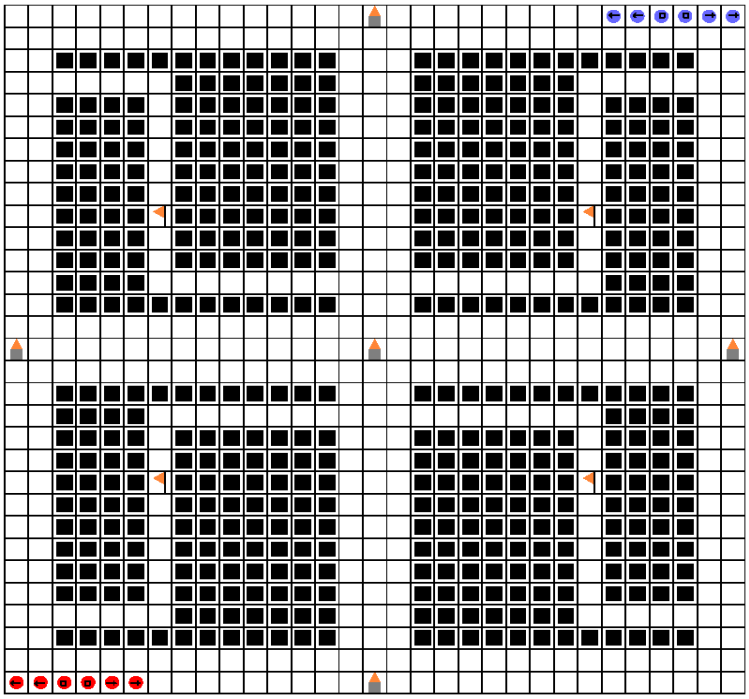
Útočící agent: Síla agenta = 5, zranění = 3
Bránící se agent: Síla agenta 4, počet životů = 2

Nejdříve je výpočítána šance *0,2.5 = 1*, tedy útok je na 100 % úspěšný. Útočící agent zasadí zranění 3, počet životů po tomto útoku tedy bude roven -1. Protože je tato hodnota <= 0, je síla branícího se agenta snížena o 1 a jeho životy nastaveny zpět na hodnotu 2. Protože ale útočící agent zasadil větší zranění než byl aktualní počet životů bránícího se agenta, je tento přesah dále odečten, tedy celkově jsou hodnoty po útoku následující:

Útočící agent: Síla agenta = 5, zranění = 3
Bránící se agent: Síla agenta = 3, počet životů = 1

3.6 Návrh testovacích modelů prostředí

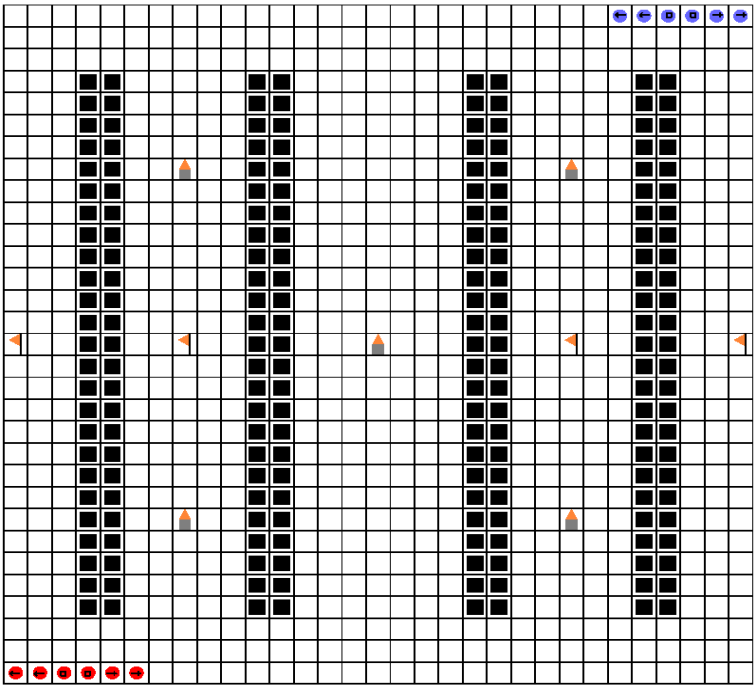
V rámci této práce byly navrženy 3 různé modely prostředí pro testování schopností různých typů chování agentů. Tyto modely jsou navrženy tak, aby rozmístění prvků a stavy v prostředí modelovaly různé situace pro řešící agenty. Detailní popis vlivu těchto prostředí na činnost agentů lze nalézt v kapitole **6**.



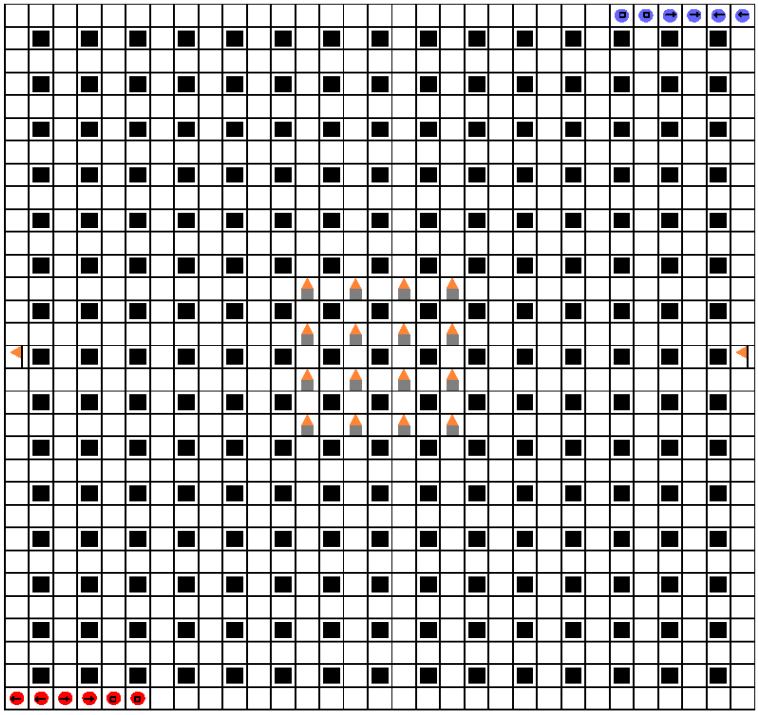
Obr.3.6: Testovací prostředí – model A.

Na obrázku **3.6** je zobrazen první testovací model – model A. Výchozí pozice každého týmu je v protilehlých rozích mřížky. Aby agenti nezačínali všichni na stejném místě, je pozice každého posunuta o `i` pozic vpravo (tým A) nebo vlevo (tým B), kde `i` je index agenta v rámci týmu. Každý tým má celkem 6 agentů, 2 od každého typu. Skóre potřebné pro výhru týmu je nastaveno na hodnotu 10000. Tato hodnota je zvolena záměrně, aby simulace netrvala příliš dlouho a zároveň aby trvala dostatečnou dobu pro porovnání efektivnosti týmů agentů.

Na obrázku **3.7** je zobrazen model B. Tento model je podobný modelu A, liší se pouze rozmístěním prvků na herní mřížce. I přesto, že vnitřní stavy prostředí jsou stejné (výchozí pozice agentů, potřebné skóre pro vítězství, typy agentů,..), změna rozmístění má vliv na efektivnost týmů. Více informací týkající se těchto odlišností lze nalézt v kapitole **6**.



Obr.3.7: Testovací prostředí – model B.



Obr.3.8: Testovací prostředí – model C.

Na obrázku **3.8** je zobrazeno poslední testovací prostředí – model C. Tento model se od modelů A a B liší kromě rozmístění i dalšímy parametry. Prostředí obsahuje větší počet věží – modely A a B obsahovaly 5 věží, model C obsahuje věží 16. Podobně je to i s vlajkami – model C obsahuje pouze 2. S větším počtem věží je spojeno i zvětšení potřebného skóre pro vítězství – z 10000 na 500 000. Výchozí pozice a počet agentů v týmu zůstává stejný jako v předchozích modelech. Dopad těchto úprav na efektivnost agentů je opět popsán v kapitole **6**, která se této problematice věnuje.

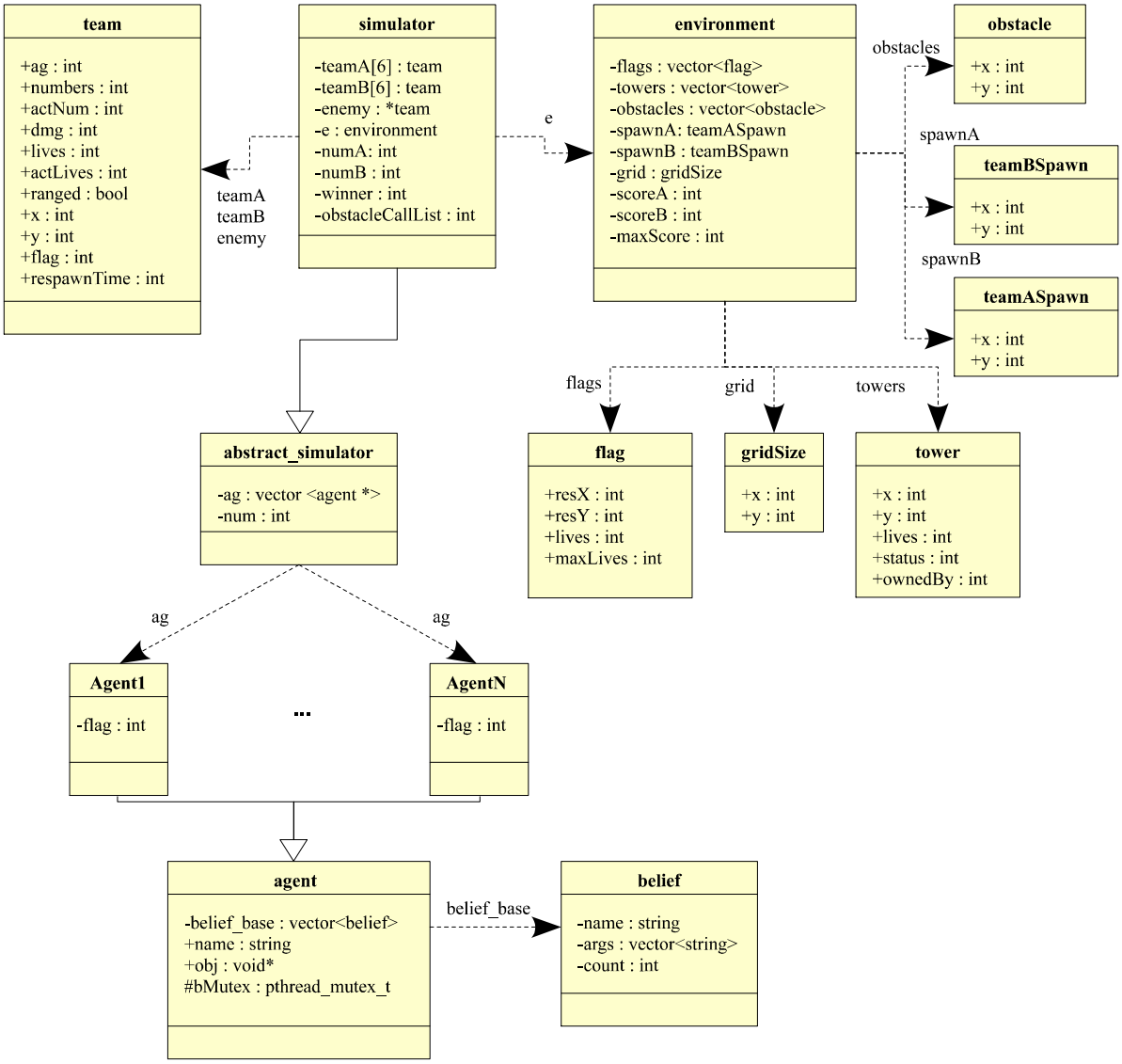
Následující tabulka shrnuje některé vlastnosti výše popsaných modelů:

	Model A	**Model B**	**Model C**
Rozměry herní plochy	31x31	31x31	31x31
Výchozí pozice týmu A	0,0	0,0	0,0
Výchozí pozice týmu B	30,30	30,30	30,30
Útoční agenti	2	2	2
Obranní agenti	2	2	2
Agenti útočící na dálku	2	2	2
Počet věží	5	5	16
Počet vlajek	4	4	2
Potřebné skóre	10000	10000	500000
Skóre za obsazené věže	$5 . 2^x$	$5 . 2^x$	$5 . 2^x$
Skóre za odnesení vlajky	$15 . 2^x$	$15 . 2^x$	$15 . 2^x$

Tab. 3.1: Shrnutí parametrů navržených modelů.

4 Implementace simulátoru multiagentního systému

V této kapitole je popsána implemetance navrhovaného simulátoru. Obrázek **4.1** zobrazuje diagram tříd implementovaného programu.



Obr. 4.1: Diagram tříd implementovaného simulátoru.

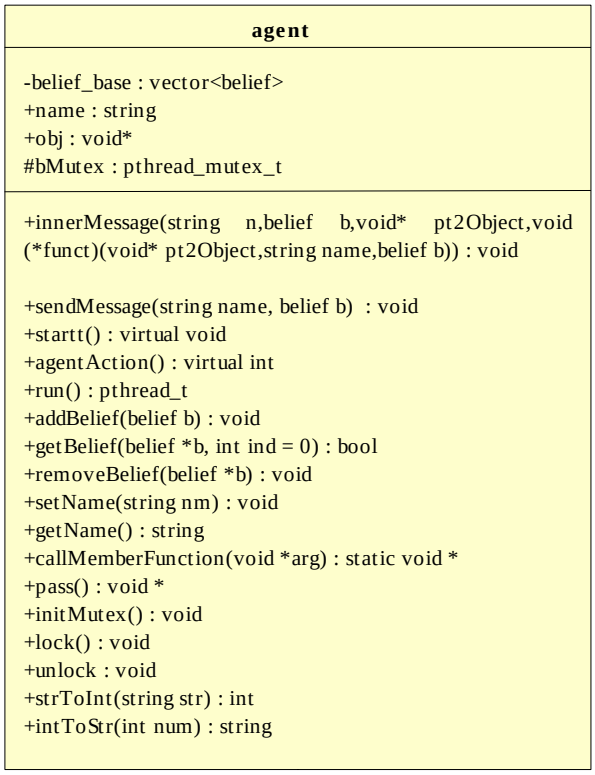
Třída `simulator` reprezentuje simulovaný model prostředí. Třídy `team` a `environment` definují parametry agentů (team) a rozmístění prvků na herní mřížce (environment). Tato třída dědí prvky z třídy `abstract_simulator`, který reprezentuje simulační rozhraní (viz kapitola **3.1**). Model prostředí má díky zděděným metodám přístup k jednotlivým agentům v simulátoru. Tito agenti jsou

reprezentováni každý svoji vlastní třídou (na obrázku pro `N` agentů jsou to třídy `Agent1` až `AgentN`). Všichni agenti dědí ze společné třídy `agent`. Tato třída implementuje společné vlastnosti agentů a řídí ovládání báze znalostí agentů. Postup provedení jednoho kroku simulace je následující:

1. `abstract_simulator` vytvoří pro všechny agenty samostatná vlákna a spustí jejich činnost.
2. Po skončení činnosti agentů je provedena úprava stavů modelu prostředí na základě akcí agentů.
3. Model prostředí pošle agentům aktualizované informace o stavu prostředí.
4. Prostředí je vykresleno na obrazovku.

4.1 Implementace agentů

Obrázek **4.2** ukazuje rozhraní třídy implementující základní strukturu agenta.



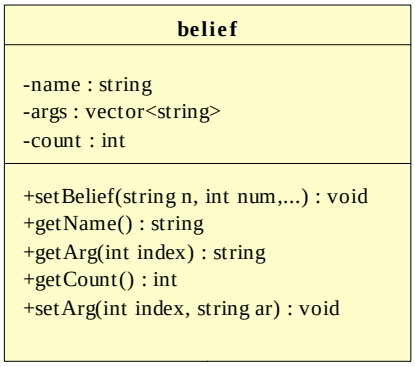
Obr. 4.2: Třída `agent`.

Třída agent obsahuje prvky společné pro každého agenta v systému:

`name` - Ukládá jméno agenta. Toto jméno slouží jako identifikátor agenta při vzájemném zasílání zpráv.
`belief_base` – Implementuje bázi znalostí agenta. Ukládá prvky typu `belief`.
`bMutex` – Mutex pro nekonfliktní přístup do báze představ.
`obj` – Ukazatel na řídící rozhraní(potřebný pro zasílání zpráv jiným agentům).

4.1.1 Třída belief

Tato třída implementuje znalost agenta. Znalost je reprezentována třemi údaji: **název znalosti (name), počet argumentů (count)** a **argumenty znalosti (args)**.



Obr. 4.3: Třída `belief`.

Metoda `setBelief` nastaví znalost na požadovanou hodnotu. Je to metoda s proměnným počtem parametrů, kde `n` je název znalosti, `num` počet argumentů (povinné parametry) a dále je seznam argumentů. Chceme-li například nastavit znalost o tom, že Petr, Jan a Jakub jsou studenti `-student(Petr,Jan,Jakub)`, zavoláme metodu `setBelief` takto:

```
belief b;
b.setBelief(”student”,3,”Petr”,”Jan”,”Jakub”);
```

Další definované metody slouží k přístupu k názvu znalosti (`getName`), argumentu (`getArg`) a počtu argumentů (`getCount`).

Poslední metoda je pomocná metoda pro přímou změnu argumentu znalosti, bez nutnosti volat metodu `setBelief`. Například pokud potřebujeme změnit `student(Petr,Jan,Jakub)` na `student(Pavel,Jan,Jakub),` můžeme zavolat `b.setArg(0,“Pavel“).`

4.1.2 Práce s bází znalostí

Báze znalostí je tedy vektor objektů typu `belief`. Agent musí mít možnost do tohoto vektoru přistupovat. Pro tyto účely má k dispozici 3 metody: `addBelief(belief b)`, `getBelief(belief *bel)` a `removeBelief(belief *bel)`.

Jelikož všechny tyto metody přistupují k prvkům vektoru, je nutné nad tímto vektorem uzamykat mutex pro výhradní přístup. V opačném případě by mohlo docházet k nekonzistenci při komunikaci dvou a více agentů (agenti jsou implementováni jako samostatná vlákna). Pro tvorbu mutexu je využita knihovna **pthread**. Pro uzamknutí/odemknutí mutexu jsou potom třídě k dispozici metody `lock()` a `unlock()`. Tyto metody zavolají funkci na zamknutí/odemknutí mutexu `pthread_mutex_lock/pthread_mutex_unlock` z knihovny **pthread**.

Metoda `addBelief(belief b)` je to metoda pro přidání znalosti do vektoru báze znalostí. Toto přidání se provede zavoláním metody `push_back` třídy vektoru, sloužící právě k těmto účelům:

```
void addBelief(belief b){
 lock();
 belief_base.push_back(b);
 unlock();
}
```

Metoda `bool getBelief(belief *bel, int ind=0)` je metoda, která prohledává bázi a snaží se najít znalost, kterou je schopna unifikovat se zadaným parametrem `bel`. Pokud nalezne vhodnou znalost, uloží ji do předávané `bel` a vrátí hodnotu **true**. Parametr `ind` určuje index představy v případě že je vhodných znalostí nalezeno více. Implicitně je tento parametr nastaven na hodnotu 0, tedy metoda vyhledá první vhodný výskyt v bázi. Pokud nenalezne vhodný záznam, proměnná `bel` se nezmění a je vrácena hodnota **false**.

Máme-li například bázi znalostí takto naplněnou (pozice nepřátelských agentů na herní mřížce):

```
enemy(1,2)
enemy(2,3)
enemy(2,2)
```

a provedeme část kódu:

```
belief b;
b.setBelief(“enemy”,2,”2”,””);
getBelief(&b);
```

Metoda vyhledá první záznam, se jménem `enemy` a prvním argumentem rovným 2 (prázdné uvozovky značí neznámou hodnotu). V tomto případě po provedení tohoto kódu bude proměnná b obsahovat záznam `enemy(2,3)`.

Metoda `removeBelief` pracuje na podobném principu jako předcházející. Vyhledá záznam, který lze unifikovat se zadanou znalostí, ale namísto úpravy hodnoty parametru metody, tento záznam z vektoru báze znalostí odstraní.

4.1.3 Metody pro spuštění agenta

Spuštění agenta je provedeno zavoláním metody `run`. Dochází k vytvoření nového vlákna, které provádí akce agenta. Pro tvorbu vláken je využita dříve zmiňovaná knihovna **pthread** (kód z této kapitoly je převzat a upraven z [7]).

```
pthread_t run(){
  pthread_t tid;
  int       result;
  result = pthread_create(&tid, 0, agent::callMemberFunction, this);
```

```
  return tid;
}
```

Vlákno je vytvořeno voláním funkce `pthread_create`. První argument je proměnná typu `pthread_t`, což je struktura obsahující informace o vlákně. Druhý argument je ukazatel na funkci, která je spuštěna při vytvoření vlákna a jejímž úspěšným skončením je vlákno zrušeno. Třetí argument je ukazatel na parametr spouštěné funkce. Funkce vrací strukturu `pthread_t` vlákna.

Jelikož funkce `pthread_create` není schopna jako vstupní funkci zpracovat metody, je nutné vytvořit statickou metodu, která zabalí skutečné volání požadované metody. Jako argument této metody je předáván ukazatel na vlastní objekt (`this`), jenž tento objekt zpřístupní. Metoda `callMemberFunction` je definována takto:

```
static void*  callMemberFunction(void *arg) {
  return ((agent*)arg)->pass();
}
```

Volání metody `pass` je poslední mezikrok před skutečným spuštěním činnosti agenta.

```
void* pass(){startt();}
```

`startt` je virtuální metoda, která je implementována ve třídě popisující chování agenta. Zavoláním `startt` dochází ke spuštení definované činnosti agenta. Po skončení volání činnost agenta končí a agent čeká na další zavolání metody `run` (v dalším kroku simulace).

4.1.4 Komunikace s okolními agenty

Agenti spolu mohou komunikovat zasíláním představ do báze znalostí. Protože přímý přístup agenta k objektu jiného agenta není možný, probíhá komunikace pomocí *callback* funkce definované v simulátoru (viz. Kapitola **4.2**). Metoda zaslání zprávy je definovaná takto (kód z této kapitoly je převzat a upraven z [8]):

```
void sendMessage(string name, belief b){
   innerMessage(name,b,obj, abstract_simulator::wrapperSendMessage);
}
```

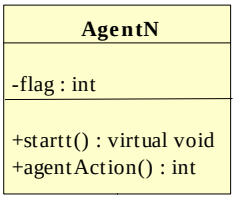
kde `name` je jméno agenta, kterému chceme poslat zprávu (znalost) a `b` je zasílaná zpráva (znalost). Dále voláme metodu `innerMessage`, kde `obj` je ukazatel na řídící rozhraní a metoda `wrapperSendMessage` je statická metoda, definovaná v simulátoru, která obaluje vlastní zaslání zprávy:

```
void innerMessage(string n,belief b,void* pt2Object,void (*funct)(void*
pt2Object, string name, belief b)){
     funct(pt2Object,n,b);
}
```

Tato metoda volá předanou `wrapperSendMessage`, která se dále stará o zaslání zprávy.

4.1.5 Implementace plánování a akcí agenta

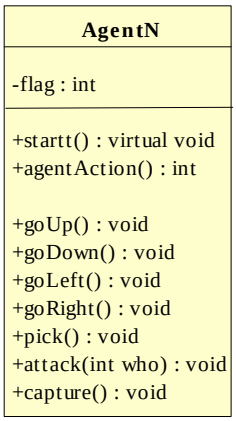
Zatímco výše popsané metody jsou společné pro každého agenta, plánování je různé pro různé typy agentů. Výše popsaná třída `agent` je děděna třídou popisující chování agenta. Instance takovéto třídy je pak již objekt plnohodnotně reprezentující agenta v systému. Kostra definice této třídy je na obrázku **4.4**:



Obr. 4.4: Kostra třídy implementující akce agenta.

Metody `startt` a `agentAction` jsou virtuální metody definované třídou `agent`. Jak bylo popsáno výše, metoda `startt` je zavolána při spuštění běhu agenta a je v ní napsána vlastní činnost agentů. `agentAction` je metoda, která vrací naplánovanou akci agenta (akce která se provede je opět konkrétní pro daného agenta). Je volána prostředím pro zjištění akce. Tento výsledný agent je přeložen jako dynamická sdílená knihovna, jenž je načítána řídícím rozhraním.

Jak bylo uvedeno v kapitole **3.5**, pro řešení simulované úlohy mají agenti k dispozici různé akce které mohou v každém kole provést. Tito agenti jsou dále definováni různými parametry, jako síla zranění a další. Pro implementaci takového agenta je třída `AgentN` rozšířena o další metody, jak ukazuje obrázek **4.5** :



Obr. 4.5: Rozšíření kostry o akce agenta.

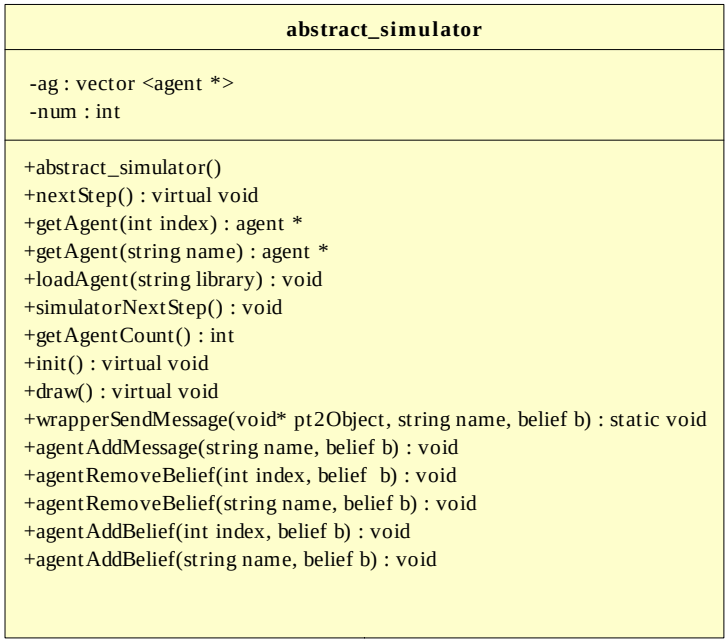
Nově přidané metody definují akce agenta tak jak byli popsány v kapitole **3.5**. Zařazení agenta do skupiny a určení jeho typu je pak možné provést v konstruktoru.

```
agentN()
{
	initMutex();
	name = “Petr“;
	belief b;
	b.setBelief("TYPE",1,"RANGED");
	addBelief(b);
	b.setBelief("TEAM",1,"A");
	addBelief(b);
}
```

Takto napsaný konstruktor vytvoří agenta se jménem *Petr* a uloží mu do báze znalostí informace o jeho typu (v tomto případě agent může útočit na dálku a je členem týmu A). Tyto informace jsou nutné pro správné zařazení agenta do týmu simulátorem modelu prostředí. Nutnou součástí inicializace agenta je i inicializace jeho mutexu pro výlučný přístup k bází znalostí. Podrobný popis implementace chování agenta lze nalézt v kapitole **5**.

4.2 Implementace simulátoru

Simulátor je definován ve třídě `abstract_simulator`, zobrazené na obrázku **4.6**:



Obr. 4.6: Implementace simulátoru.

Proměnná `ag` je vektor ukazatelů na vytvořené agenty. Pomocí této proměnné je možné přistoupit k metodám agenta. Proměnná `num` obsahuje počet vytvořených agentů v simulátoru.

4.2.1 Načtení agentů

Načtení agenta se provede metodou `loadAgent(string library)`, kde `library` je cesta k dynamické sdílené knihovně obsahující přeloženého agenta.

```
typedef agent* create_t();
typedef void destroy_t(agent*);
void loadAgent(string library){
    void* agentt = dlopen(library.c_str(), RTLD_LAZY);
    if (!agentt) {
        cout << "nenacteno" << dlerror();
    }
    dlerror();
    create_t* create_agent = (create_t*) dlsym(agentt, "create");
    const char* dlsym_error = dlerror();
    if (dlsym_error) {
        cout << "nenalezeno" << dlsym_error;
    }
    destroy_t* destroy_agent =(destroy_t*) dlsym(agentt, "destroy");
    ag.push_back(create_agent());
    ag[num]->obj = this;
    num++;
}
```

Pro načtení funkcí ze sdílené knihovny je využita knihovna **dlfcn**. Tato knihovna umožňuje načítat přeložené dynamické sdílené knihovny za běhu programu. V tomto případě je použita pro načtení funkcí, které vytvoří a zruší instanci objektu agenta.

Načtení knihovny se provede funkcí `dlopen` s parametrem cesty k dynamické knihovně. Voláním `dlsym` jsou vytvořeny funkce `create_agent` a `destroy_agent`. Funkce `dlsym` prohledá načtenou dynamickou knihovnu (první argument) a hledá funkci s názvem zadaným druhým argumentem (v tomto případě je to “create“ a “destroy“). Tyto funkce jsou definované v souboru spolu s chováním agenta. Například pro agenta uvedeného v kapitole **4.1.5** jsou to:

```
extern "C" agent* create() {
    return new agentN();
}

extern "C" void destroy(agent* p) {
    delete p;
}
```

Funkce `create` vrací novou instanci objektu `ag1` (agenta). `destroy` naopak istanci ruší. Klíčovým slovem `extern "C"` je nutné překladači explicitně sdělit, že se nejedná o metody, ale o funkce (překladač předpokládá funkce jako metody nějaké třídy). Bez této úpravy by překladač nebyl schopen korektně přeložit zdrojový soubor do sdílené knihovny.

```
ag.push_back(create_agent());
ag[num]->obj = this;
num++;
```

Funkcí `create_agent` je vytvořena instance agenta, uložena do vektoru agentů v řídícím rozhraní a inkrementována hodnota, značící počet načtených agentů. Po načtení je nutné agentovi předat ukazatel na řídící rozhraní `obj` (k volání callback funkce pro zasílání zpráv).

4.2.2 Operace s agenty

Komunikační rozhraní obsahuje metody pro přístup k jeho agentům. Jedná se zejména o metody přístupu k bázi znalostí agenta. Tyto metody jsou však jenom mezivrstva pro zjednodušení přístupu k agentům z prostředí. Prostředí má možnost přímo vyhledat agenta na základě jeho jména, nebo podle jeho pořadí uložení ve vektoru, a získat přímého přístupu k agentovi a jeho bázi znalostí.

● **Přístup k bázi znalostí** - Jak již bylo řečeno, jedná se pouze o mezivrstvu mezi agentem a prostředím. Například metoda pro přidání znalosti je definována:

```
void agentAddBelief(int index, belief b){
     getAgent(index)->addBelief(b);
}
```

`getAgent` získá ukazatel na agenta na pozici `index`. Nad tímto agentem se zavolá jeho metoda pro přidání představy `addBelief`. Ostatní operace (odebrání a získání znalosti) jsou definované podobným stylem

● **Metody pro přístup k agentovi** - Řídící rozhraní nabízí metody pro získání ukazatelů na jeho agenty:

```
agent* getAgent(int index)
{
      if (ag[index] != NULL)
       return ag[index];
      else
       return NULL;
}
```

Jednoduchá metoda, která vrátí ukazatel na agenta z pole, z indexu zadaného jako argument. Kromě této metody je dále definována:

```
agent* getAgent(string name)
{
          int j = -1;
          for (int i = 0; i < num;i++)
          {
              if ((ag[i]->getName()).compare(name) == 0) j=i;
          }
          if (j == -1)return NULL;
          agent *agA = ag[j];
          return agA;
}
```

Tato metoda vyhledává agenta na základě jeho jména. Vyhledávané jméno je typu `string` a je předáno jako argument metody. Metoda porovnává všechny jména všech agentů se zadaným, a nalezne-li shodu, vrátí tohoto agenta.

- **Callback funkce pro meziagentní komunikaci**

```
static void wrapperSendMessage(void* pt2Object,string name,belief b)
{
     abstract_simulator* mySelf = (abstract_simulator*) pt2Object;
     mySelf->agentAddMessage(name,b);
}
```

Pro *callback* volání metod je nutné metodu definovat jako statickou. Do této metody je předán jako argument ukazatel na vlastní objekt. Pomocí tohoto ukazatele je dále zpřístupněna metoda `agentAddMessage`, která uloží předanou představu `b` do báze představ agenta se jménem `name`.

4.2.3 Virtuální metody pro implementaci prostředí

Řídící rozhraní poskytuje modelu prostředí metody, které model prostředí může implementovat.

`nextStep -` Simulace jednoho kroku v prostředí.
`init` – Inicializace prostředí.
`draw` – Vykreslení prostředí na obrazovku.

`nextStep` je jediná metoda, kterou je nutné v modelu prostředí implementovat. Jejím voláním řídící rozhraní spustí simulaci kroku pro prostředí. U ostatních metod záleží na návrhu prostředí a úlohy (není potřebná inicializace nebo vykreslení prostředí).

4.2.4 Provedení kroku simulace

Krok simulace celého systému je proveden zavoláním `simulatorNextStep`:

```
void simulatorNextStep()
{
  pthread_t tids[num];
  for (int i = 0; i < num;i++)
  {
    tids[i] = ag[i]->run();
  }
  for (int i = 0; i < num;i++)
  {
    pthread_join(tids[i], NULL);
  }
  nextStep();
}
```

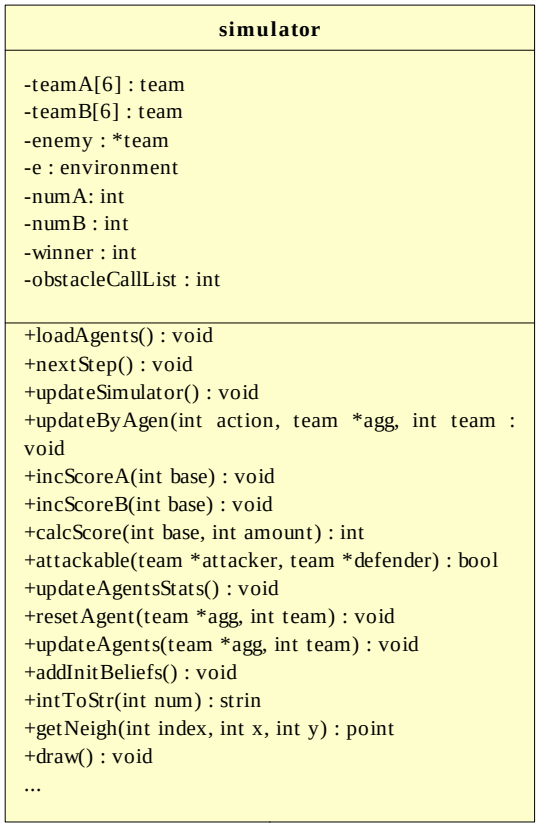
Řídící rozhraní pro všechny své agenty zavolá metodu `run`, čímž spustí vlákna agentů a nechá je pracovat. Informace o vláknech si uloží do pomocného pole. Pro zjištění, kdy agent

ukončil svou činnost (ukončilo se vlákno), se využívá funkce `pthread_join` z knihovny **pthread**. Tato funkce zablokuje činnost hlavního procesu (ze kterého byly vytvořeny vlákna), dokud vlákno zadané prvním argumentem neukončí činnost.

Po ukončení všech vláken (všichni agenti skončili svoje plánování a zvolili svou akci) je spuštěn krok simulace prostředí. Opakovaným voláním `simulatorNextStep()` je prováděna vlastní simulace celého systému.

4.3 Implementace modelu prostředí

Tato kapitola obsahuje popis implementace prostředí, ve kterém se agenti pohybují. Třída dědí metody a data ze simulátoru (třídy `abstract_simulator`).



Obr. 4.7: Třída modelu prostředí.

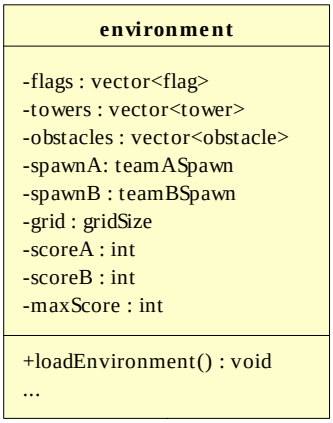
Proměnné ve třídě simulátor představují vnitřní stavy modelu prostředí, případně pomocné proměnné pro provádění simulace. Jedná se o tyto proměnné:

`teamA` – Pole obsahující údaje o agentech z týmu A.
`teamB` – Pole obashující údaje o agentch z týmu B.
`enemy` – Pomocný ukazatel na nepřítele při útocích agenta.
`e` – Proměnná obsahující rozvržení objektů v mřížce. Jedná se o proměnnou typu `environment`.

numA – Velikost pole `teamA`.
numB – Velikost pole `teamB`.
winner – Proměnná obsahující informaci zda vyhrál některý tým.
obstCallList – *callList* pro vykreslování překážek.

4.3.1 Rozvržení umístění objektů na herní mřížce

Proměnná typu `environment` (viz obrázek **4.8**) obsahuje informace o umístění objektů v prostředí a nastavení pravidel úlohy.

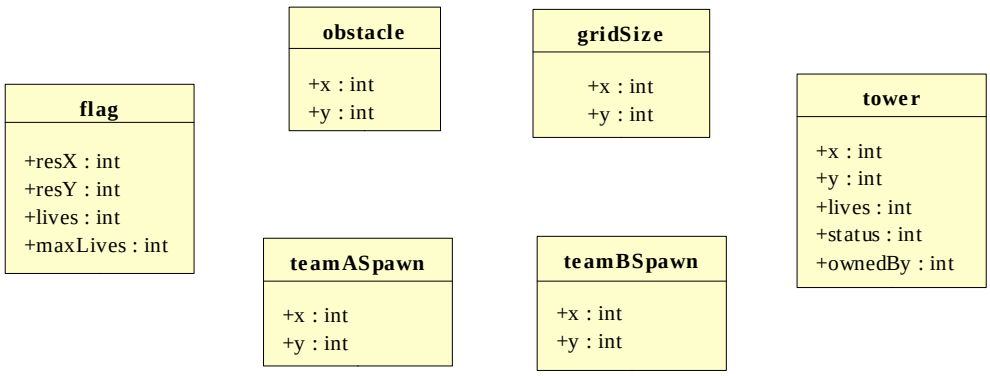


Obr. 4.8: Reprezentace rozmístění objektů na herní mřížce.

Třída `environment` obsahuje tyto proměnné:

flags – Popis všech vlajek ve hře.
obstacles – Popis překážek.
towers – Popis věží.
spawnA – Startovní pozice agentů z týmu A.
spawnB – Startovní pozice agentů z týmu B.
gridSize – Velikost herní mřížky.
scoreA – Skóre týmu A.
scoreB – Skóre týmu B.
maxScore – Cílové skóre.

Datové typy těchto proměnných jsou struktury, zapouzdřující parametry, patřící k příslušným objektům. Tyto struktury plně popisují objekt v modelu prostředí. Jednotlivé parametry lze nalézt na obrázku **4.9**.



Obr. 4.9: Struktury obsahující parametry jednotlivých prvků prostředí.

- **Struktura `flag`**

Obsahuje informace o vlajce v prostředí. Proměnná `maxLives` představuje maximální počet životů vlajky, kterou agenti musí použít pro její úspěšné sebrání. Proměnné `resX` a `resY` obsahují umístění vlajky v prostoru herní mřížky. Proměnná `lives` obsahuje aktuální počet životů vlajky po pokusu o sebrání.

- **Struktura `obstacle`**

Obsahuje údaje o pozici překážky na herní mřížce. Tyto údaje jsou uloženy v proměnných `x` a `y`.

- **Struktura `tower`**

Obsahuje informace o věži v prostředí. Proměnná `lives` je počet životů věže po obsazení některým z týmů. Proměnná `status` je aktuální počet životů po pokusu o obsazení. Proměnné `x` a `y` obsahují umístění věže v prostoru herní mřížky. Proměnná `ownedBy` obsahuje údaj o aktuálním vlastníkovi věže.

- **Struktury `spawnA` a `spawnB`**

Struktury `spawnA` a `spawnB` obsahují informace o místě, kde se agent přesune po jeho smrti a kde je umístěn na začátku simulace. Tyto informace jsou uloženy v proměnných `x` a `y`.

- **Struktura `gridSize`**

Obsahuje rozměry herní mřížky. Proměnná `x` určuje počet řádků a proměnná `y` počet sloupců.

Třída environment načítá rozmístění z externího hlavičkového souboru. Pro tyto účely má definovánu metodu `loadEnvironment()`:

```
void loadEnviroment(){
  env env1;
  flags = env1.loadFlags();
  towers = env1.loadTowers();
  obstacles = env1.loadObstacles();
```

```
  gridSize = env1.getGridSize();
  spawnA = env1.getASpawn();
  spawnB = env1.getBSpawn();
  maxScore = env1.getMaxScore();
}
```

Rozmístění překážek je definováno ve třídě `env`. Tato třída obsahuje výše použité metody, které vytvoří vektory a proměnné s požadovaným rozmístěním a parametry a vrátí je jako návratovou hodnotu. Metoda `loadEnvironment()` postupně volá tyto metody a jejich výsledky uloží do svých datových struktur. Tento způsob načítaní prvků prostředí umožňuje snadnou změnu parametrů a rozmístění prvků prostředí. Stačí načíst jiný soubor s definovanou třídou `env,` ze které se načtou nové vektory a proměnné.

Třída `environment` obsahuje dále metody pro přístup k jednotlivým proměnným a jejich atributům (z důvodu jejich vysokého počtu nejsou na obrázku **4.8** uvedeny). Pomocí těchto metod je model prostředí schopen upravovat svoje stavy a tím simulovat celý úkol.

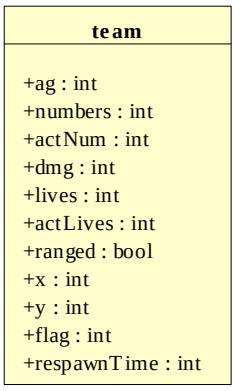
4.3.2 Inicializace prostředí

Inicializace prostředí probíhá ve zděděné virtuální metodě `init()`. Postup inicializace je následující:

1. Do proměnné `environment e` je načteno rozmístění prvků a parametry prostředí (`e.loadEnvironment()`).
2. Je vytvořen *callList* pro vykreslování prostředí (kapitola **4.4**).
3. Agenti jsou načteni do prostředí (kapitola **4.3.3**).
4. Inicializace generátoru náhodných čísel.
5. Agentům jsou uloženy do báze znalostí základní údaje obsahující informace o prostředí (viz kapitola **3.4**).

4.3.3 Načtení agentů do prostředí

Prostředí prohledá všechny vytvořené agenty v simulátoru a uloží si o nich údaje do proměnných typu `team` - `teamA[]` (pro tým A) a `teamB[]` (pro tým B).



Obr. 4.10: Struktura s informacemi o agentech.

`Proměnná ag` je index agenta v simulátoru, `numbers` a `actNum` je velikost (síla) týmu, `dmg` je velikost zranění, `lives` a `actLives` je počet životů, `ranged` značí zda agent může útočit na dálku, `x` a `y` je pozice na herní mřížce, `flag` označuje zda a jaká vlajka je agentem nesena a `respawnTime` je doba po kterou agent není schopen jednat po jeho zlikvidování v boji.

Načtení agentů do prostředí probíhá v metodě `loadAgents()` podle následujícího algoritmu:

1. Postupně v cyklu načti všechny agenty v systému (počet agentů vrací metoda `getAgentCount`).
2. Získej příslušnost k týmu agenta z údaje v jeho bázi znalostí TEAM`(tym)`.
3. Získej typ agenta z údaje v jeho bázi znalostí `TYPE(typ)`.
4. Nastav parametry korespondující struktury reprezentující parametry agenta (`team`) podle získané znalosti `TYPE` (konkrétní hodnoty lze nalézt v kapitole **3.5**). `respawnTime` je inicializován na hodnotu 0 (agent je naživu), `flag` na hondotu -1 (agent nemá žadnou vlajku), počáteční pozice je načtena z proměnné typu `environment`, kde je tato hodnota definována a do proměnné `ag` je uložen index agenta.
5. Inkrementuj proměnnou určující počet agentů v týmu (`numA`, případně `numB`).

4.3.4 Provedení kroku simulace prostředí

Provedení kroku simulace modelu prostředí je definováno ve zděděné virtuální metodě `nextStep`. Činnost této metody lze popsat následujícím pseudoalgortimem:

1. Kontrola, zda je simulace skončena (`winner != -1`). Pokud ano, metoda je ukončena bez provedění jakýchkoli změn v prostředí.
2. Změna stavů modelu prostředí a parametrů agentů na základě provedených akcí agenta v tomto kroku simulace.
3. Kontrola, zda některý z týmů zvítězil. Podle toho je nastavena proměnná `winner` na hodnotu 0 (zvítězil tým A) nebo 1 (zvítězil tým B).
4. Vymazání neaktuálních informací o stavu prostředí z bází znalostí agentů a nahrazení aktuálními informaci.

Při změně stavů modelu na základě akce agenta je pro každého agenta nejprve zjištena hodnota proměnné `respawnTime`. Pokud je rovna 0, agent je aktivní a spustí se metoda pro změnu stavů na základě akce vybraného agenta. Pokud je větší než 0, znamená to, že agent byl v předchozích kolech zabit. V tomto případě je proměnná dekrementována o 1 a zpracování akce tohoto agenta je přeskočeno. Podle zjištěné akce agenta je proveden jeden z následujích pseudoalgoritmů :

- **Pohyb agenta**

```
změn souřadnice agenta podle směru;
if(agent je mimo plochu || agent je na překážce)
  vrať se na původní pozici;
```

Jedná se o pohyb agenta zadaným směrem. Pohyb je proveden přičtením nebo odečtením hodnoty `1` k souřadnicím agenta (podle směry pohybu). Po této změně je nutné zkontrolovat, zda se touto akcí agent nedostal mimo herní mřížku, nebo nevstoupil na překážku. Pokud ano, jsou souřadnice vráceny zpět na původní hodnoty a metoda končí.

- **Sebrání vlajky**

```
if (na souřadnici agenta je vlajka){
      odečti sílu agenta od životů vlajky;
      if (síla vlajky <= 0 && agent nenese vlajku){
            ulož agentovi index vlajky;
            ulož agentovi do báze informaci o nesení vlajky;}}
```

Na souřadnicích agenta je vyhledána vlajka. Nenachází-li se na těchto souřadnicích, je metoda ukončena. V opačném případě je od počtu životů vlajky odečtena hodnota síly týmu agenta. Klesne-li počet životů vlajky na 0 nebo méně a agent nenese jinou vlajku, je nastavena hodnota parametru `flag` agenta na číslo vlajky (vlajka je sebrána). Agentovi je do báze znalostí zaslána informace o držení vlajky `CARRYING(Jméno_agenta)`.

- **Položení vlajky**

```
if (agent se nachází na věží && vlastní věž && nese vlajku){
      vrať počet životů vlajky na maximální hodnotu;
      index vlajky agenta nastav na -1;
      odeber z báze informaci o nesení;
      inkrementuj skóre týmu agenta;
}
```

Nejprve je získána věž na souřadnici agenta. Zkontroluje se, zda taková věž existuje, vlastní ji tým agenta a agent nese vlajku. Pokud ano, vlajka je vrácena zpět na své místo s plným počtem životů a agentovi je z báze odstraněna informace o držení vlajky. Týmu agenta, který ji odesl, je inkrementováno skóre podle zadaného vzorce.

- **Žádna akce/zabírání věže**

```
if (agent se nachází na věži && nevlstní věž){
      odečti/přičti k životů věže sílu agenta;
      if (počet životů věže překročil hodnotu 0){
            nastav počet životů na maximum;
            nastav nové vlastnictví věže;
      }
}
```

V případě, že agent nechce provést žádnou akci, je automaticky proveden pokus o zabrání věže. Po ověření existence věže na pozici agenta je k životům věže přičtena (v případě týmu A), nebo odečtena (tým B) síla týmu agenta. Je-li hodnota počtu životů kladná (tým A), nebo záporná (tým B) ,je nastaveno vlastnictví věže agentovi a její životy jsou nastaveny na maximální hodnotu (kladnou v případě týmu A, zápornou v případě B).

- **Útok**

```
do pomocné proměnné enemy ulož parametry agenta podle zadaného indexu;
if (nepřítel je v dosahu útočícího agenta && útok se podařil){
      odečti od počtu životů bránícího se agenta velikost zranění;
      while (počet životů <= 0){
            sniž sílu nepřátelského agenta o 1;
            přičti maximální počet životů k aktuálnímu počtu životů;
      }
}
```

Není-li prováděna žádná z předchozích akcí, znamená to, že agent útočí. Do pomocné proměnné `enemy` je uložen ukazatel na nepřátelského agenta. Následuje zjištění, zda na nepřátelského agenta je možné útočit (agent musí být na stejném políčku jako útočící, v případě agentů s útokem na dálku i na sousedním). Dále je vypočítáno, zda bude útok úspěšný na základě pravděpodobnosti úspěchu.

Při úspěšném útoku je odečtena od životů bránícího se agenta velikost zranění útočícího agenta. Klesne-li počet životů pod 0, je dekrementována síla bránícího se agenta a jeho životy jsou zvýšeny o hodnotu jeho maximálních životů. Toto se opakuje do té doby, dokud je počet životů menší než 0.

Jelikož agenti mohou na sebe útočit vzájemně a útoky probíhají současně, je třeba údaje o počtech ubraných životů a snížené síly agenta ukládat do dočasných proměnných. Po skončení zpracování činnosti všech agentů je zavolána metoda, která dočasné hodnoty přepíše na skutečné hodnoty agenta. Zjistí-li tato metoda, že některý agent po přepsání údajů má sílu 0 (byl zabit), vrátí agenta na počáteční pozici, vrátí zpět vlajku kterou případně nesl a nastaví mu `respawnTime` na 20 kol simulace.

Po provedení změn prostředí na základě akcí agenta je inkrementováno skóre hry. S tím je spojená kontrola, zda některý z týmů dosáhl požadovaného skóre a zvítězil. Pokud ano, je nastavena proměnná `winner` podle vítěze.

Poslední část provedení kroku simulace spočívá v aktualizování báze znalostí agentů. Následující seznam popisuje informace které jsou agentům odstraněny z báze a nahrazeny údaji novými:

Pos (x, y) – pozice agenta.
Numbers (num) – síla agentovi skupiny.
ScoreA (score) – skóre týmu A.
ScoreB (score) – skóre týmu B.
Enemy (x, y, index) – pozice nepřátelského agenta, nacházejícího se v bezprostředním okolí od agenta a jeho index.
TowerPos (x, y, index,ownage) – pozice věže informace od vlastnictví věže, index věže a informace o vlastníkovi věže.
NearFlag (x, y,index) – pozice vlajky v bezprostředním okolí od agenta a její index.
Obstacle (x, y) – pozice překažky v bezprostředním okolí od agenta.
FriendPos(name, type, x, y) – pozice přátelských agentů, jejich jméno a typ

4.4 Spuštění simulace a vykreslení na obrazovku

Pro vizualizaci modelu prostředí je použita knihovna **OpenGL** (http://www.opengl.org/) a **glut** (http://www.opengl.org/resources/libraries/glut/). Knihovna **OpenGL** slouží pro práci s grafikou (2D i 3D). Pro zjednodušení práce s touto knihovnou je použita knihovna **glut** (*OpenGL utility toolkit*)**.** Základní funkce poskytovaná touto knihovnou je vytvoření a nastavení okna aplikace nezávisle na použité platformě:

```
int main(int argc, char **argv){
  ////////////////////////////////
  //inicializace a nastavení glut///
  ////////////////////////////////
  glutDisplayFunc(onDisplay);
  glutIdleFunc(wait);
  Init();
  glutMainLoop();
  return 0;
}
```

Inicializace a nastavení je blok, ve kterém probíhá tvorba okna podle zadaných parametrů a jsou nastaveny vlastnosti okna. Dále probíhá registrace funkcí reagujících na různé události systému (např. klik myši nebo stisk klávesy). Z hlediska implementace simulátoru jsou důležité dvě funkce – `glutDisplayFunc`, která registruje funkci volanou při překreslení okna aplikace a `glutIdleFunc,` která je volána v době, kdy není aktuálně zpracovávána jiná událost a slouží k simulaci systému. Funkce `glutMainLoop` představuje hlavní smyčku **glut** aplikace.

4.4.1 Inicializace programu

Je definován ve funkci `Init`:

```
abstract_simulator *s;
void Init(){
  //inicializace OpenGL knihovny
  s = new simulator;
  s->loadAgent("./libAgAR0.so");
  s->loadAgent("./libAgBD0.so");
  s->init();
}
```

Jak je znázorněno na obrázku **4.1**, simulátor se vytvoří jako instance modelu prostředí, dědící vlastnosti z řídícího rozhraní. Do simulátoru jsou načteni agenti (zde pro ilustraci jsou to dva agenti z knihoven *libAgAR0.so* a *libAgBD0.so*). Po načtení agentů je zavolána metoda `init`, která inicializuje prostředí (viz kapitola **4.3.2**).

4.4.2 Krokování simulace

Ke krokování simulace je použita funkce `wait`. Tato funkce je zaregistrována funkcí `glutIdleFunc`, jako funkce, která je volána pokaždé, kdy není prováděna činnost žádné jiné funkce **OpenGL**.

```
void wait(){
  usleep(delay);
  nextStep();
}

void nextStep(){
  s->simulatorNextStep();
  glutPostRedisplay();
  return;
}
```

Na začátku spuštění jednoho kroku simulace je pozdrženo provádění programu na dobu danou hodnotou `delay` (v mikrosekundách). Změnou této hodnoty lze řídit rychlost simulace. Po vypršení doby čekání je proveden krok simulace systému (kapitola **4.2.4**) a překreslení okna.

4.4.3 Vykreslení prostředí

Funkcí `glutDisplayFunc` je zaregistrována funkce, kterou **OpenGL** volá při požadavku na vykreslení scény. Pro účely vykreslení stavu simulace je v modelu prostředí implementována metoda `draw`. V této metodě probíhá vykreslení různých prvků systému:

- **Herní mřížka –** Vykreslení herní mřížky.
- **Překážky –** Vykreslení překážek na odpovídající pozice na herní mřížce.
- **Věže** – Vykreslení věží na odpovídající pozice na herní mřížce , obarvené podle vlastníka věže.
- **Vlajky** – Vykreslení vlajek na odpovídající pozice na herní mřížce za předpokladu, že je žádný agent nenese.
- **Agenti** – Vykreslení agentů na odpovídající pozice na herní mřížce, obarvené podle týmu. K agentům je dále nakreslen symbol identifikující, zda agent nese vlajku a symbol určující jeho typ.
- **Skóre hry** – Vykreslení aktuálního skóre hry pro oba týmy agentů.

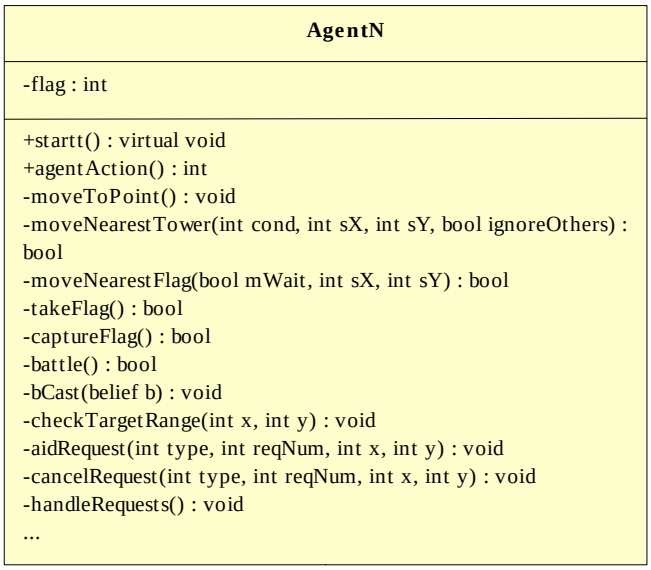
Pro více informací o vykreslování objektů a programování pomocí knihoven **OpenGL** a **glut** lze nalézt v [9] a [10].

5 Tvorba agentů

Tato kapitola obsahuje popis implementace činnosti jednotlivých agentů. Jak bylo popsáno v kapitole **4.1.5**, základní třída obsahující činnost agenta vypadá takto:

```
class ag1 : public agent{
public:
  virtual void startt();
  int agentAction();
};
```

kde `startt` je metoda která je volána simulátorem a obsahuje výpočet akce agenta a `agentAction` je metoda, která tuto akci vrací. Pro účely prostředí a úlohy zvolené pro tuto práci je tato třída rozšířena podle obrázku **5.1**.



Obr. 5.1: Rozšíření základní třídy reprezentující chování agenta.

Každému agentovi, vytvořeného podle této třídy, je nutné na začátku uložit do báze představ potřebné počáteční údaje. Tato inicializace je provedena v konstruktoru této třídy. Jedná se o tyto údaje:

`TEAM(TYP_AGENTA)` – Určuje typ agenta, kde `TYP_AGENTA` nabývá hodnot DMG (útočný agent), DEF (obranný agent) a RANGED (agent útočící na dálku).
`TYPE(TEAM)` – Určuje tým do kterého agent náleží. `TEAM` nabývá hodnot A (tým A) a B (tým B).
`DIRECTION(DIR)` – Určuje momentální směr pohybu (vlevo, vpravo, nahoru nebo dolu). Význam tohoto údaje bude vysvětlen dále.
`DIRTO(DIR1, DIR2)` – Podobně jako předchozí, určuje směr správné cesty, význam bude vysvětlen dále.
`OWNED(0)` – Obsahuje počet zabraných věží agenta (na začátku každý tým 0).

`ACTTARGET(-1, -1)` – Pozice na mřížce aktuálního cíle, kterého se agent snaží dosáhnout (na začátku není zatím zvolen žádný cíl).

Celkem je vytvořeno 12 agentů, 6 pro tým A a 6 pro tým B. Třídy těchto agentů jsou pojmenovány tak, aby bylo poznat o kterého agenta se jedná. Seznam agentů lze nalézt v tabulce **5.1**.

Jméno třídy	Jméno agenta	Tým	Typ	Priorita
agAR1	Agent1	A	Na dálku	4
agAR2	Agent2	A	Na dálku	5
agAD1	Agent3	A	Útočný	2
agAD2	Agent4	A	Útočný	3
agAO1	Agent5	A	Obranný	0
agAO2	Agent6	A	Obranný	1
agBR1	Agent7	B	Na dálku	4
agBR2	Agent8	B	Na dálku	5
agBD1	Agent9	B	Útočný	2
agBD2	Agent10	B	Útočný	3
agBO1	Agent11	B	Obranný	0
agBO2	Agent12	B	Obranný	1

Tabulka 5.1: Seznam agentů v prostředí.

Znaky ag ve jméně třídy značí, že se jedná o agenta. Další písmena určují tým a typ. Poslední znak je k rozlišení dvou agentů stejných typů. Hodnota priorita z tabulky je vysvětlena v kapitole **5.2**.

5.1 Metody pro plánování činnosti

Všichni agenti, bez rozdílu jejich typu a týmu, mají implementovány několik metod, které plánují vhodnou činnost agenta. Jedná se o **pohyb po mřížce**, **vyhledání nejbližší věže**, **vyhledání nejbližší vlajky**, **sebrání vlajky**, **odevzdání vlajky** a **boj**.

5.1.1 Pohyb po mřížce

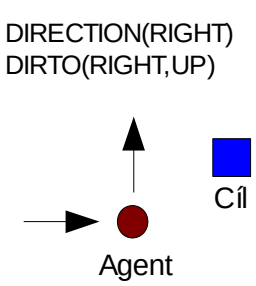
Pro pohyb agentů po mřížce je použita metoda `moveToPoint(int x, int y)`, kde `x` a `y` značí pozici kam se agent snaží přesunout. Funkce vrací hodnotu `true` v případě, že je možný pohyb některým směrem a `false`, v případě že se na cílové políčko nelze dostat (cíl leží na překážce nebo mimo herní mřížku).

Vyhledávání cesty funguje tak, že na začátku je uložena do báze informace o směru, kterým se agent, bez ohledu na překážky, dostane do cílového místa. Je-li agent například na pozici (5,10) a cílová pozice je (8,30), je do báze uloženo `DIRTO(RIGHT,UP)` (tedy agent se dostane k cíli půjde-li doprava a nahoru, souřadnice (0,0) začínají v levém dolním rohu). Agent si dále uchovává v záznamu

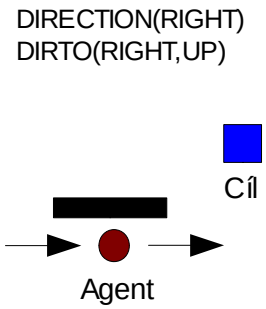
`DIRECTION(DIR)` směr, kterým se posunul v minulém kole (při spuštění programu nebo při změně cílové pozice je nastavena na RIGHT nebo LEFT). Algortimus nalezení vhodného směru pohybu vypadá následovně:

1. Pokud se agent nachází na cílovém místě nedělej nic.
2. Pokud je možné změnit směr agenta o 90° na jeden ze směrů určených v `DIRTO`, jdi tímto směrem.
3. Pokud je možné pokračovat směrem, kterým agent šel poslední tah (`DIRECTION`), jdi tímto směrem
4. Pokud lze změnit směr agenta o 90° na směr, který není uložen v `DIRTO`, jdi tímto směrem.

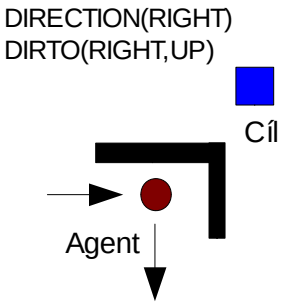
Takto navržený výpočet směru pohybu, umožňuje agentovi vyhýbat se překážkám, na které během své cesty narazí. Následující obrázky znázorňují výše zmíněné možnosti.



Obr. 5.2: Krok 2, změna směru o 90°.



Obr 5.3: Krok 3, pokračování v původním směru.



Obr 5.4: Krok 4, změna do opačného směru než v obr. 5.2.

5.1.2 Vyhledání nejbližší věže

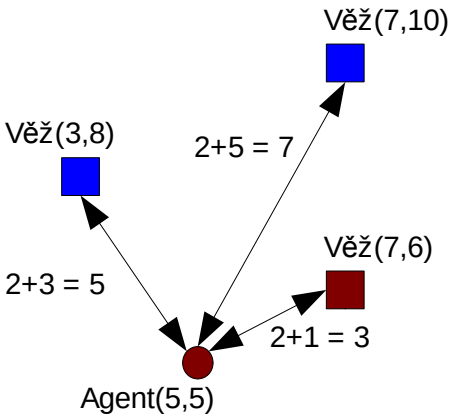
Pro nalezení nejbližší věže od agentova je použita metoda `bool moveNearestTower(int cond, int sX, int sY, bool ignoreOthers)`. Vzdálenost je vypočítána jako součet absolutních hodnot rozdílu souřadnic agenta a cíle. Například, je-li agent na pozici (5,10) a cílová pozice je (8,30), je vzdálenost |5 - 8| + |10 - 30| = 23. Hodnoty parametrů `sX` a `sY` jsou implicitně nastaveny na hodnotu -1. Jsou-li tyto parametry zadány jinou (platnou) hodnotou, jsou dosazeny do vzorce pro výpočet vzdálenosti místo pozice agenta.

Argument `cond` je filtr který určuje věže, které bude metoda brát v úvahu. Nabývá těchto konstantních hodnot:

`NOBODY` – uvažuje věže které nikdo nevlastní
`TEAM_A` – věže které vlastní tým A
`TEAM_B` – věže které vlastní tým B

Tyto hodnoty je možné kombinovat pomocí bitového operátoru OR. Například zavoláním `moveNearestTower(NOBODY | TEAM_B)` způsobí, že agent vyhledá nejbližší věž, kterou

nikdo nevlastní, nebo ji vlastní tým B. Pokud je nalezena věž, která splňuje zadané podmínky, je zavolána výše popsaná metoda `moveToPoint` pro přesun k této věži a je vrácena hodnota `true`. Pokud vhodná věž není nalezena, metoda vrací `false`. Argument `ignoreOthers` slouží k ignorování akcí od ostatních agentů. Popis spolupráce s ostatními agenty lze nalézt v kapitole **5.2**.



Obr. 5.5: Ukázka výběru cílové běže.

Obrázek **5.5** ukazuje rozhodování agenta při výběru vhodné věže. Za předpokladu že volání metody nespecifikuje žádný filtr, agent vybere nejbližší věž, tedy věž na pozici (7,6), která má hodnotu vzdálenosti od agenta 2. V případě, že volání metody filtruje vežě, které agent vlastní, je vybrána další nejbližší věž, která splňuje zadaný filtr, tedy v tomto případě je to věž na pozici (3,8).

5.1.3 Vyhledání nejbližší vlajky

Pro nalezení nejbližší vlajky od agenta je použita metoda `bool moveNearestFlag(bool mWait, int sX, int sY)`. Výpočet vzdáleností probíhá stejně jako při hledání věží. Při vyhledávání vlajek je však nutné počítat s tím, že vlajku mohl sebrat jiný, například nepřátelský agent. O tom, že vlajka se na svém místě nenachází, se agent dozví, až je v její bezprostřední blízkosti. Nastevení argumentu `mWait` na `true` způsobí, že pokud agent dorazí na pozici nejbližší vlajky, a vlajka chybí, počká na tomto místě, dokud se vlajka neobnoví. Je-li argument nastaven na `false`, agent na místě nečeká a vydá se hledat další nejbližší vlajku. Vynechána z vyhledávání je i vlajka, která byla sebrána jiným přátelským agentem (viz kapitola **5.2**). Metoda, stejně jako při hledání věží, volá `moveToPoint` pro přesun směrem k nalezené vlajce.

5.1.4 Sebrání vlajky

Implementováno v metodě `bool takeFlag()`. Metoda zkontroluje nachází-li se agent na políčku s vlajkou a nenese už jinou vlajku. Pokud ano, provede sebrání vlajky a vrací true. V opačném případě vrací `false`.

5.1.5 Odevzdání vlajky

Implementováno v metodě `bool captureFlag()`. Metoda zkontroluje, nachází-li se agent na políčku s obsazenou věží a nese vlajku. Pokud ano, provede odevzdání vlajky a vrací `true`. V opačném případě vrací `false`.

5.1.6 Boj

Implementováno v metodě `battle()`. Tato metoda zjistí možnost a vhodnost útoku na jiného agenta. V případě, že metoda zjistí, že agent má zaútočit, zavolá funkci pro útok a vrací `true`. V opačném případě vrací `false`. Implementace této metody se liší v závislosti na typu agenta - agent útočící na dálku ji implementuje jinak než ostatní agenti. Podrobnější popis implementace lze nalézt v kapitole **5.3**.

5.2 Spolupráce agentů

U týmu agentů bez jakékoliv spolupráce často dochází k tomu, že celý tým generuje stejnou akci. Takový tým nemusí být příliš efektivní – agenti nevyužívají všechny prvky prostředí rovnoměrně. Z hlediska možného zvýšení efektivity agentů je důležité, aby agenti o sobě věděli a dokázali spolu komunikovat. Agenti používají dva způsoby komunikace: **Informování o aktualním cíli** a **Zasílání pomocných informací**.

5.2.1 Informování o aktuálním cíly

Jedná se o způsob, jakým agent informuje ostatní o jeho aktuálním cíli a o zpracování takto zaslané informace. Každý agent má v bázi uloženy jména všech ostatních agentů v týmu a informaci o jejich počtu (tyto informace jsou zaslány prostředím před začátkem simulace). Z využitím těchto informací je implementována metoda `bCast`. Metoda `bCast` provádí broadcast údaje zadaného argumentem `b` všem agentům. Informace o počtu agentů je uložena v `NUMF(cnt)` a jednotlivá jména v `FRIEND(name)`. Metoda v cyklu projde všechny záznamy `FRIEND` a pro každé jméno zašle přes `sendMessage` zprávu obsahující argument `b`. Tento argument má formát `FRIENDACT(name,type,x,y)`, kde `name` je jméno zasílajícího agenta, `type` je jeho typ (DMG, DEF nebo RANGED) a `x,y` jsou souřadnice aktuálního cíle agenta. Kromě této znalosti agent může poslat `INVALIDATE(name)`, kde `name` je jeho jméno. Tato zpráva značí, že agent změnil cíl a přijemci zprávy mají předchozí `FRIENDACT` zprávu ignorovat a smazat. Tento princip zasílání zpráv je využit u vyhledávání věží a vlajek.

- **Zpracování údajů o agentů**

U vyhledávání věží, agent pro každou pozici věže, kromě vzdálenosti zjistí, zda jeho báze neobsahuje záznam `FRIENDACT` se shodnými souřadnicemi. Pokud ano, tak podle priority agenta (různé typy agentů mají různé priority) buď vež vynechá ze zpracování (nižší priorita), nebo pokračuje beze změn (vyšší priorita). Je-li argument `ignoreOthers` nastaven na `true`, metoda pracuje se všemi

věžemi, bez kontroly cílů ostatních agentů. U vyhledávání vlajek funguje stejný princip jako u věží. Existuje-li záznam v bázi, který značí že jiný agent má za cíl stejnou vlajku, je podle priority určeno, který agent pokračuje ve směru vlajky a který mění cíl.

Změní-li přátelský agent svůj cíl, zašle nový údaj `FRIENDACT` spolu s `INVALIDATE`. Každý agent na začátku nového kroku kontroluje, zda jeho báze obsahuje záznam `INVALIDATE`. Pokud ano, odstraní jej (aby nebyl načten v dalším kole) spolu se všemi záznamy `FRIENDACT` od určeného agenta, kromě posledně přidaného (aktuálního cíle).

- **Výpočet aktuálního cíle**

Výpočet aktuálního cíle je prováděn v metodě `checkTargetRange(int x, int y)`. Tato metoda je volána z metod pro výpočet nejbližší věže nebo vlajky před samotným pohybem k vybrané věži/vlajce. Argumenty metody jsou vypočítané souřadnice cílové věže, případně vlajky. Tyto souřadnice jsou porovnány se záznamem `ACTTARGET`, který obsahuje souřadnice aktuální pro minulý krok. Pokud jsou souřadnice stejné, znamená to, že agent v tomto kroku nezměnil cíl a metoda je ukončena. Jsou-li rozdílné, starý záznam vymaže a nahradí jej aktuálním. Tyto souřadnice jsou dále metodou `inform` zaslány pomocí výše definované metody `bCast` všem agentům jako aktuální cíl (metoda `inform` zabalí souřadnice do specifikované představy spolu s dalšími potřebnými údaji jako je jméno a typ agenta). Nakonec je broadcastována zpráva INVALIDATE, značící ostatním agentům že mají zneplatnit předchozí údaje.

5.2.2 Zasílání pomocných informací

Jedná se o způsob komunikace, kdy agenti zasílají požadavky, případně další doplňující informace jiným agentům. Každý přijmutý požadavek je agentem zpracován – buď je přijmut, nebo odmítnut. Dále zaléží na implementaci činnosti agenta, jestli se rozhodne přijmutý požadavek vykonat, nebo pokračuje ve své vlastní činnosti.

Zaslání požadavku provede agent voláním metody `aidRequest(int type, int reqNum, int x, int y)`. Podle agrumentu `type` se rozlišují dva typy požadavků: `BATTLE_AID` a `GUARD_AID`.

`BATTLE_AID` - Jedná se o požadavek o pomoc při útoku nebo obraně na nějakou pozici na mřížce. Tato pozice je zadaná argumenty `x` a `y`. Argument `reqNum` udává kolika agentům je požadavek zaslán. Metoda vybere `reqNum` nejbližších agentů a zašle jim zprávu `BattleReq(name,x,y)`, kde `x` a `y` je předaná pozice, kam má agent dojít útočit a `name` je jméno zasílajícího agenta.

`GUARD_AID` - Je požadavek o obranu věže z důvodu volného prostoru pro položení vlajky žádajícím agentem, případně zablokování věže, aby nepřátelský agent vlajku na tuto věž nemohl odnést. Argumenty `x` a `y` udávají pozici, odkud má agent vycházet při vyhledávání nejbližší věže (jako by se na ní nacházel) a `reqNum` obsahuje filtr pro vyhledávání věží. Metoda vybere nejbližšího agenta obranného typu a zašle mu zprávu `GuardReq(name,x,y,reqNum)`.

Pokud se agent rozhodne, že jeho požadavek už není důležitý, může pomocí metody `cancelRequest(int type, int reqNum, int x, int y)` broadcastovat zneplatňující zprávu (type značí jaký typ požadavku zneplatňuje).

Kromě těchto požadavků mají agenti dále možnost poslat informační údaj o tom, že agent sebral, případně položil vlajku. Jedná se o údaje `FlagTaken(name,flagInd)` pro informování že sebral vlajku, kde `name` je jméno agenta a `flagInd` je index vlajky a `FlagDropped(name)` pro informování že vlajka byla položena.

- **Zpracování požadavků**

Na začátku každého kroku agent zavolá metodu `handleRequests()`, která zpracuje výše popsané požadavky a informace.

`BATTLE_AID` - Agent postupně v cyklu vyhledá všechny `BattleReq` záznamy. Pokud nějaký nalezne (jiný agent zaslal požadavek), zjistí zda už nemá přijatý jiný požadavek. Pokud ne, požadavek přijme – uloží si do báze údaj `AID(BATTLE, name, x, y)`, kde `BATTLE` identifikuje typ požadavku, `name` je jméno agenta který požadavek poslal a `x` a `y` jsou zaslané souřadnice. Všechny `BattleReq` jsou během tohoto cyklu odstraněny z báze.

Další údaj, který agent zpracovává je požadavek na zrušení `CancelBattleReq`. Tento požadavek je zpracován podobným způsobem jako `BattleReq`, ale místo vytvoření `AID` záznamu v bázi, všechny `AID` záznamy se jménem odesilatele vymaže.

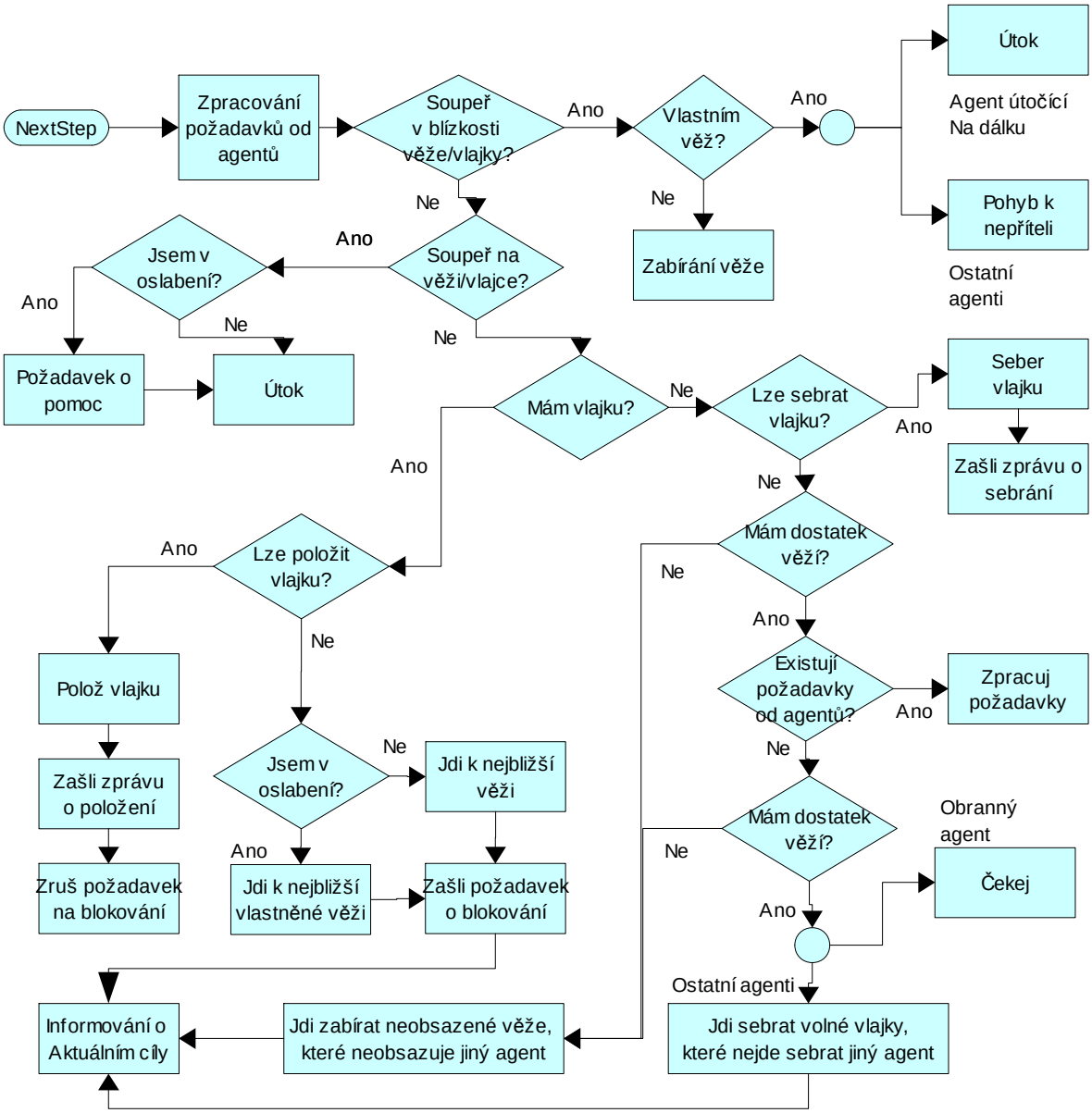
`GUARD_AID` - V případě zpracování `GuardReq` je situace složitější. V předcházejícím případě má agent uložen záznam o tom, že má jít útočit na zadanou souřadnici a všechny další požadavky ignoruje, dokud na zadanou pozici nedojde. V případě `GuardReq` musí souřadnice cíle aktualizovat pokaždé, kdy mu od zadaného agenta přijde nový, jelikož agent který požaduje pomoc se z různých důvodů může rozhodout že změní svůj cíl (viz kapitola **5.3**). Zpracování probíhá podobně jako v případě `BattleReq`. Agent si uloží do báze představ údaj `AID(GUARD, name, x, y, filter)`, kde `filter` obsahuje informaci pro filtrování při hledání nejbližší věže, zbytek parametrů je stejný jako v předchozím případě. V případě že už tento záznam existuje, je nahrazen novým pouze tehdy, je-li jméno nového požadavku shodné se jménem `name` v `AID`. V opačném případě je chování identické se zpracováním `BattleReq`. Stejné je i zpracování `CancelGuardReq` požadavku.

FlagDropped - Jako další je zpracován záznam `FlagDropped`, kterým byl agent informován, že vlajka zadaná jako parametr je položena a vrácena zpět na své místo. V tomto případě je tento záznam z báze odstraněn, spolu s `FlagTaken`, který značil že vlajka byla sebrána.
Kromě těchto požadavků je ve funkci `handleRequest` zpracován ještě jeden údaj - `DIED(DIED)`. Tento údaj je uložen do báze agenta modelem prostředí v situaci, kdy agent zemřel a znovu se objevil na výchozí pozici. V tomto případě agent broadcastuje zprávu `FlagDropped`

(pokud byl zabit a nesl vlajku, musí dát vědět ostatním agentům že ji již nenese) a `cancelRequest(GUARD_AID, …)` (stejný případ, dává vědět agentovi, že požadavek na obranu už není aktualní, protože nenese vlajku).

5.3 Vytvoření umělé inteligence agentů

S využítím metod z kapitol **5.1** a **5.2** je implementována inteligence agenta v prostředí. Veškeré zde popisované metody jsou použity pro agenty v týmu A. Metody pro tým B jsou velmi podobné, liší se pouze odkazy na týmy (např. při zabírání věží). Agentova inteligence se liší typem agentů. Každý typ agenta se bude chovat jinak než jiné typy, aby co nejvíce využil svých vlastností a zamaskoval své nevýhody. Agenti jsou implementování ve třech různých úrovních spolupráce. Diagram na obrázku **5.6** zobrazuje pseudoalgoritmus činnosti agenta:



Obr.5.6: Diagram činnosti agentů.

Zpracování požadavků od agentů je provedeno voláním metody `handleRequests()`, která, podle postupu v kapitole **5.2** zpracuje příchozí požadavky od agentů.

Následujících několik bloků reprezentují implementaci metody `battle()`. Pokud je soupeř v blízkosti věže/vlajky, pro agenta to znamená možné ohrožení. Pro agenta je v tuto chvíli prioritou věž zabrat, aby generovala týmu body. Pokud však agent má věž obsazenou, zaútočí na tohoto blízkého soupeře. V tomto místě je implemetance odlišná pro různé typy agentů – agent útočící na dálku může přímo zaútočit, ostatní agenti musí nejdřív přejít na soupeřovu pozici.

Pokud se soupeř nachází přímo na věži/vlajce, na které se nachází i agent, je proveden útok na agenta bez ohledu na vlastnictví věže. Před útokem je spočítána síla agenta vůči nepříteli – pokud je agent v oslabení(`num > 0`) oproti nepříteli, zašle pomocí `aidRequest(BATTLE_AID, num, x, y)` požadavek o pomoc v boji. Parametr `num` je vypočítán podle intenzity oslabení – `num = enemies + isWeak – 1`, kde `enemies` je počet nepřátel a `isWeak = 1`, pokud má agent sílu skupiny < 3, jinak `isWeak = 0`.

Další část algortimu se dělí podle toho, zda agent nese nebo nenese vlajku. Pokud nese, je v jeho zájmu ji co nejrychleji odevzdat. Nachází-li se agent na obsazené věži, vlajku odevzdá a rozešle všem agentům příslušné zprávy – `FlagDropped` o tom že vlajka byla položena a `CancelGuardReq` o zrušení požadavku na blokování věže obranným agentem. Pokud se na vlastněné veži nenachází, zavolá metodu `moveNearestTower(filtr,true)` pro přesun k nejbližší věži a `aidRequest(GUARD_AID, filtr, x, y)`. Agent v této fázi odlišuje zda je vážně zraněn nebo ne. Pokud je, filtruje z vyhledaných věží všechny nepřátelské (jejich obsazení před položením vlajky by ho zbytečne zdrželo a riskoval by smrt od nepřítele). Argument true je nutný proto, aby byly brány v potaz opravdu všechy nevyfiltrované věže, nejenom ty, které nemají za cíl jiní agenti.

Pokud agent nenese vlajku, ale nachází se na pozici s vlajkou, sebere ji a současně zašle všem agentům zprávu `FlagTaken`. Zbylá část algoritmu se liší podle typu agenta a sestává se z následujících kroků:

1. Pokud má tým agenta méně jak určitý počet obsazených věží (< 3 útočný, < 2 obranný, < 0 na dálku), agent volá `moveNearestTower` s filtrováním obsazených věží (jde zabírat nevlastěné věže) a s filtrováním věží, které už jde obsadit jiný agent s vyšší prioritou.

2. V této fázi algoritmu agent jde provádět přijmuté požadavky, má-li nějaké. V případě požadavku o bojovou pomoc `AID(BATTLE, name, x, y)` agent volá metodu pro přesun `moveToPoint(x,y)`. Pokud se pozici `x` a `y` nachází, smaže údaj AID a pokračuje dále. Pokud se jedná o požadavek blokování věže `AID(GUARD, name, x, y, filtr)`, agent volá `moveNearestTower(filtr, x, y)`.

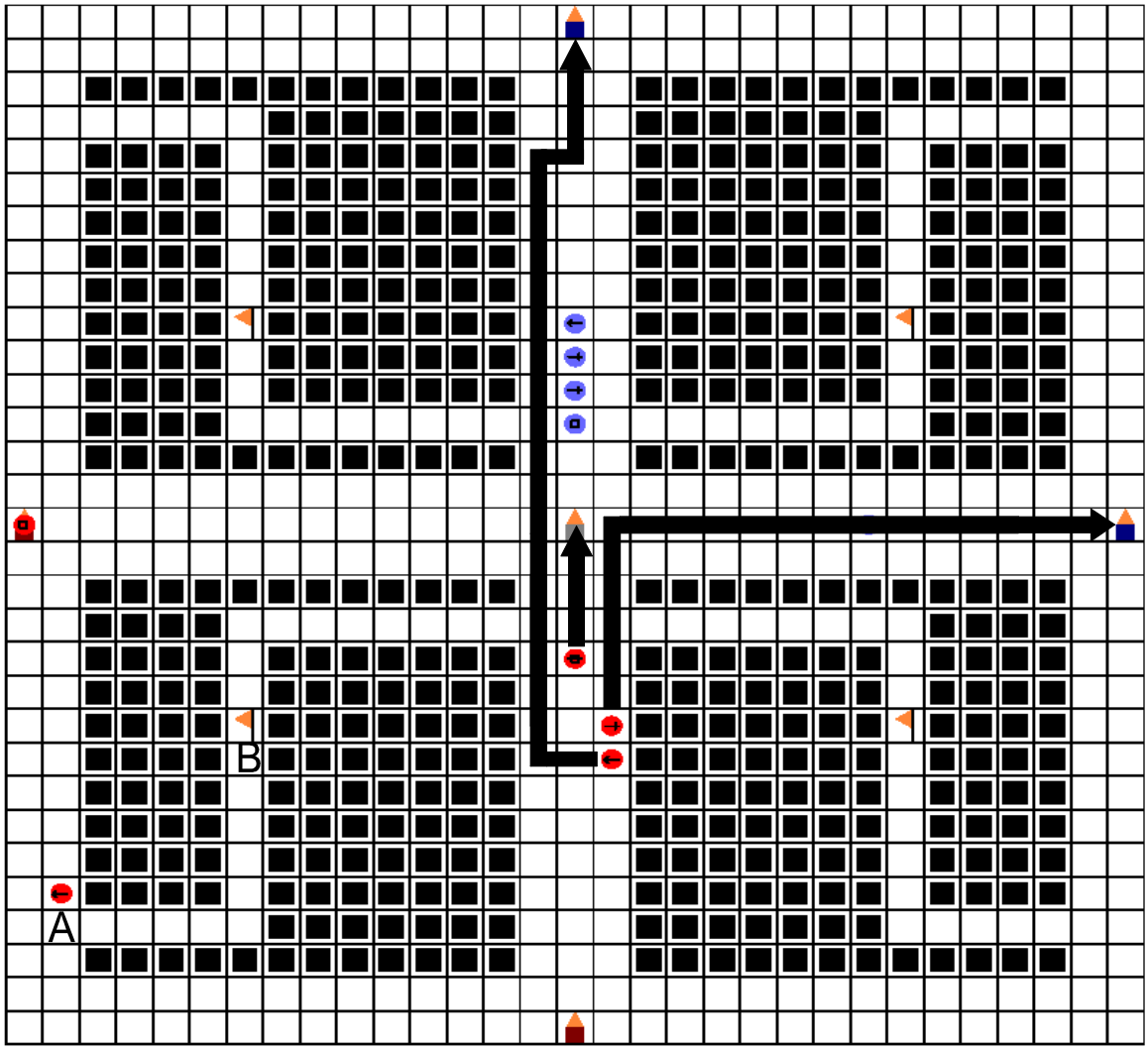
3. Pokud agent nemá žádné přijaté požadavky, provádí další kontrolu počtu obsazených věží, ale s jinými hodnotami (<= 3 útočný, <= 2 obranný, <= 2 na dálku) a chová se stejně jako v bodě 1.

4. V případě, že obsazených věží je dostatek, agenti volají `moveNearestFlag(true)` (útočný a na dálku), nebo `moveNearestTower(filtr)` (obranný, filtr dovolí pohyb jenom k obsazeným věžím, agent tedy jde bránit již obsazenou věž).

Metoda `moveNearestFlag` obsahuje ještě jednu zajímavou vlastnost z hlediska vylepšení inteligence agentů. Pokud agent dojde na místo kde očekával vlajku, která tam není, znamená to, že ji sebral nepřítel. V takovém případě agent zasílá `FlagTaken`, čímž se pro ostatní agenty vlajka tváří, jako by byla sebrána přátelským agentem a mohou ji vynechat z vyhledávání.

5.4 Příklad činnosti jednoho kroku agenta

Uvažujme model prostředí simulovaného systému zobrazený na obrázku **5.7**:



Obr. 5.7: Ukázka simulovaného systému.

Agenti jsou zobrazeni jako červené (tým A) a modré (tým B) kolečka. Symbol uprostřed zobrazuje jejich skupinu. Šipky vedoucí z některých agentů do věží znamenají, že tito agenti mají jako momentalní cíl nastaven tuto věž. Činnost agenta označeného písmenem A bude probíhat následovně (předpokládejme, že agent A neobdržel žádné požadavky od ostatních agentů a agent A je typu útočícího na dálku):

1. Agent volá metodu `battle`. Jelikož se okolo agenta nenachází žádný nepřítel, metoda vrací `false` a agent pokračuje dál.
2. Agent zjišťuje že nenese vlajku, pokusí se tedy jednu sebrat. Protože se však na jeho pozici vlajka nenachází, metoda pro sebrání vlajky vrací `false` a agent pokračuje dál.
3. Agent zjišťuje, zda má dostatek věží. Na obrázku má tým A obsazeny dvě věže, tedy je splněna podmínka pro minimální počet věží a agent pokračuje dál
4. Agent kontroluje, zda jeho báze představ neobsahuje nějaké požadavky od jiných agentů. Jelikož žádné neobsahuje, agent pokračuje dál.
5. Probíhá druhá kontrola minimálního počtu věží. Podmínka v této fázi není splněna, agent tedy spustí metodu pro vyhledání nejbližší neobsazené věže. Protože však na všechny neobsazené věže už směřují jiní agenti, metoda vrací `false` a agent pokračuje dál.
6. Agent volá metodu pro vyhledání nejbližších vlajek. Protože žádná z vlajek není již cílem jiných přátelských agentů, je vrácena nejbližší vlajka (na obrázku označena písmenem B).
7. Agent volá metodu `moveToPoint` pro přesun směrem k vlajce.

6 Provedené experimenty

V této kapitole je provedeno vyhodnocení několika různých přístupů agentů k řešení zadané úlohy. Pro účely testování byly vytvořeny tři různé modely prostředí. Popisy jednotlivých modelů lze nalézt v kapitole **3.6**. Implementace chování agentů je použita tak, jak je popsána v kapitole **5**, s několika rozdíly, lišícími se podle typů přístupu k řešení úlohy:

- **Spolupracující agenti**

Agenti mají implementovány všechny metody popsané v kapitole **5**.

- **Částečně spolupracující agenti**

Agenti mají implementovány všechny metody, kromě metod pro zasílání pomocných informací z kapitoly **5.2**. Tento způsob tedy zachovává informování o cíli ostatních agentů, ale agenti si nejsou schopni zasílat pomocné informace o sebraných vlajkách a nemohou požádat ostatní agenty o pomoc.

- **Agenti bez spolupráce**

Agenti nemají žádné metody pro komunikaci s ostatními agenty (viz kapitola **5.2**).

Zvolené modely mají kromě vizuálních odlišností i odlišnosti v logice řešení úlohy:

Model A

Model A je zobrazen na obrázku **3.6**. Rozmístění prvků v tomto protředí je zvoleno takovým způsobem, že věž, která je umístěna ve středu herní mřížky, je téměř ze všech klíčových pozic na mřížce nejblíže položená. Například, zabere-li agent některou z ostatních věží, je další nejbližší věž právě věž na středu. Podobně je to i s vlajkami, kde se často stává že krajní blízké věže jsou zabrané jiným týmem a agent vyrazí směrem na střed. Střední věž na mřížce se tedy stává jakýmsi centrem bojů agentů a její obsazení a úspěšná obrana je klíčová pro výhru týmu. Jak je ukázáno v kapitole **6.2,** existence takového místa má značný vliv na efektivnost jednotlivých typů chování agentů.

Model B

Druhé testovací prostředí je zobrazeno na obrázku **3.7**. Jak je z obrázku vidět, překážky v tomto prostředí jsou rozmístěny takovým způsobem, že pro přesun mezi jednotlivými částmi, které překážky rozdělují, je nutné tyto překážky obejít, což způsobí určité zdržení agenta. Na rozdíl od modelu A, věž na středu již není centrem bojů agentů. To je způsobeno jednak nutností obcházení zdí z překážek, jednak koncentrací většiny ostatních objektů po stranách mřížky. Tento způsob rozmístění má na efektivnost agentů určitý vliv, podobně jako v případě modelu A.

Model C

Třetí prostředí pro testování je zobrazeno na obrázku **3.8**. Tento model obsahuje větší množství věží než předchozí dva modely. Rozmístění věží na obrázku je vhodné pro testování chování agentů bez složitého obcházení (model B) nebo koncentrací agentů na jedno určité místo (model A). Toto

rozmístění způsobí, že agenti provádí činnost na menším prostoru, a dochází mezi nimi k častým soubojům.

Jak bude vidět v kapitole **6.1**, velmi významný podíl na výsledku má, kromě rozmístění obejktů v modelu, prvek náhody.

6.1 Týmy se shodnými vlastnostmi

Tato kapitola obsahuje porovnání týmů agentů, kteří mají shodný přístup k řešení úlohy. Tyto výsledky nejsou, z hlediska porovnání různých přístupů, příliš zajímavé, ale ukazují některé vlastnosti implementovaných modelů.

- **Spolupracující agenti**

Tabulka **6.1** zobrazuje výsledky 10 běhů dvou soupeřících týmů spolupracujících agentů (hodnoty v tabulce reprezentují dosažené skóre týmu). Je vidět že ve všech třech testovaných modelech je poměr výhra : prohra téměř 50 procent a průměrné dosažené skóre podobné u obou týmů. Jelikož agenti bojující proti sobě mají implementovánu stejnou úrověn spolupráce, má na situaci velký vliv náhodnost modelu prostředí. Tato náhodnost spočívá ve výpočtu šance na zasažení zranění soupeři (viz kapitola **3.5.5**). Je vidět, že výsledky ve všech běžích simulace jsou podobné, s několika vyjímkami. Například v běhu číslo 7, v Modelu B, vyhrál tým A s více něž dvounásobným náskokem před soupeřem. V této situaci se díky náhodě podařilo týmu A vyhrát větší část soubojů a obsadit rychle většinu vlajek. Tomuto odpovídá i doba (počet tahů) simulace, která je asi o 50 tahů kratší než doba průměrná. Další podobná situace je v běhu 5 v modelu C.

		Model A			**Model B**			**Model C**	
	Tým A	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**
1.	10060	8460	307	10010	9970	320	509220	187330	217
2.	6250	10010	256	9860	10000	299	352180	501270	178
3.	10010	7410	280	10010	7930	291	452860	531330	146
4.	8230	10010	293	7790	10030	276	514300	150430	137
5.	9900	10030	318	9160	10010	305	500260	53090	105
6.	10010	7470	274	7820	10010	278	520740	164850	148
7.	10000	8470	285	10010	4250	224	500660	90250	163
8.	10020	7230	279	10010	6650	255	257500	500620	215
9.	8600	10000	299	6780	10050	254	500580	137730	172
10.	7420	10040	287	9230	10000	273	158660	509330	184
Průměr	**9050**	**8913**	**287,8**	**9068**	**8890**	**277,5**	**426696**	**282623**	**166,5**
Počet výher	**5**	**5**		**4**	**6**		**6**	**4**	

Tabulka 6.1: Spolupracující agenti.

- **Částečne spolupracující agenti**

U souboje částečně spolupracujících agentů dopadla situace podobně jako v případě spolupracujících. Poměr výhra : prohra se blíží 50 procentům, průměrné skóre každého týmu je podobné. V tabulce **6.2** lze opět najít případ, kdy se některému z týmu podařilo na začátku simulace zvítězit ve většině soubojů a tak získal rychlý nárůst bodů (běh 10 v modelu C). Z hlediska průměrného počtu tahů je zajímavé, že těmto agentům trvalo v modelu B průměrně o 20 tahů déle zvítězit, než spolupracujícím agentům.

		Model A			**Model B**			**Model C**	
	Tým A	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**
1.	10000	9650	298	10010	7890	295	502820	183050	131
2.	9170	10030	300	8210	10010	297	504020	343170	180
3.	10020	7700	271	10010	9080	306	355860	512530	132
4.	10020	8760	276	9150	10000	312	500820	362450	211
5.	9880	10000	308	8630	10030	307	285870	525390	150
6.	10030	8850	285	7630	10000	272	114190	505710	149
7.	10120	9700	306	10000	9310	317	354510	504470	176
8.	9490	10050	286	7880	10020	284	554950	104390	142
9.	8120	10010	272	10000	8470	273	534390	251030	222
10.	10010	8990	278	10040	8307	287	83510	513870	117
Průměr	**9686**	**9374**	**288**	**9156**	**9311,7**	**295**	**379094**	**380606**	**161**
Počet výher	**6**	**4**		**5**	**5**		**5**	**5**	

Tabulka 6.2: Částečně spolupracující agenti.

- **Agenti bez spolupráce**

Tabulka **6.3** zobrazuje souboj dvou týmu bez spolupráce. Z tabulky je opět vidět podobný scénář jako v předchozích dvou případech. Rozložení výher sice není úplně rovnoměrné (naprř. v modelu A vyhrál tým A z 10 pokusů 8x), ale průměrné skóre je u obou týmů opět velmi podobné. Za povšimnutí v tomto měření stojí průměrný počet tahů potřebný pro vítězstvá týmu. To, že nespolupracující agenti nedokážou zjistit aktuální cíl ostatních agentů způsobuje, že všichni agenti často (nejčastěji na začátku simulace) zvolí za cíl stejné místo na herní mřížce. Toto způsobí určité zdržení, které je potom promítnuto v celkovém průmeru počtu tahů. Z tabulky jde vidět, že největší změna je v případě modelu C. To je způsobeno větším počtem věží v tomto modelu koncentrovaných na středu plochy – spolupracující agenti rychleji zaberou většinu věží, což díky exponenciálnímu nárůstu skóre způsobí rychlejší vítězství.

		Model A			Model B			Model C	
	Tým A	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**
1.	10020	9920	310	8270	10000	304	479850	503050	356
2.	10000	8770	301	10010	7750	315	288650	501930	293
3.	10040	7840	292	10000	9240	318	516490	302250	292
4.	10070	7650	281	10030	8820	317	238570	502570	216
5.	10020	9290	309	10030	7310	303	500810	323850	312
6.	10000	9070	308	10010	7590	309	223210	503850	280
7.	8780	10010	317	8720	10050	311	505450	212330	272
8.	10000	6920	293	8990	10010	331	352650	500650	201
9.	9220	10030	310	10030	8390	303	392970	502090	262
10.	10010	9320	324	10000	8480	324	500490	182410	265
Průměr	**9816**	**8882**	**304,5**	**9609**	**8764**	**313,5**	**399914**	**403498**	**274,9**
Počet výher	**8**	**2**		**7**	**3**		**4**	**6**	

Tabulka 6.3: Agenti bez spolupráce.

6.2 Týmy s rozdílnými vlastnostmi

Tato kapitola obsahuje výsledky simulací při porovnávání agentů s rozdílnými vlastnostmi. Na rozdil od kapitoly **6.1**, která ukázala, že výherce simulace je závisý z velké části na náhodě, zde bude popsán vliv prostředí na konkrétní typy chování agentů.

- **Spolupracující agenti x Částečně spolupracující agenti**

První porovnávanou skupinou agentů jsou **Spolupracující agenti** (tým A) a **Částečně spolupracující agenti** (tým B). Výsledky měření deseti běhů simulace jsou zobrazeny v tabulce **6.4**.

Jak lze z tabulky vidět, výsledky simulace v modelu A i v modelu B jsou velmi podobné výsledkům z předchozí kapitoly. Počet výher a průměrně dosažené skóre jsou si blízké i přesto, že se jedná o dva odlišné typy chování agentů. Důvod, proč tomu tak je, spočívá ve velké vzdálenosti prvků na herní mřížce mezi sebou. V takovém případě se zasílání pomocných správ často neuplatní. Například (uvažujme model B) zašle-li agent požadavek o pomoc obsazení věže, kterou brání jiný agent, tento agent často musí obejít zeď vytvořenou z překážek. Než se tento agent dostane ke věži, může nastat jiná situace a agent změní cíl jako reakce na tuto změnu, případně se dostane na cílové místo pozdě. V případě modelu A je situace taková, že vzhledem k soustředění činnosti agentů okolo centrálního místa, postrádájí požadavky efektivnost, protože oba týmy jsou v centru dění již přítomni.

V případě modelu C je situace jiná. Agenti zde plně využijí své schopností zasílat požadavky, čímž získají velkou převahu nad soupeři při zabírání věží. Jak ukazuje tabulka **6.4**, Spolupracující agenti zvítězili 9x. Částečně spolupracující agenti sice 1x zvítězili (prvek náhody v soubojích je přítomný ve

všech bojích), ale vítězství bylo velmi těsné (běh č.5). V ostatních případech zvítězil tým A s velkým náskokem nad týmem B, což lze pozorovat i na průměrném zisku skóre u obou týmů.

		Model A			Model B			Model C	
	Tým A	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**
1.	9470	10010	325	9890	10000	308	513700	22390	70
2.	10030	6250	262	9940	10010	318	504020	67450	93
3.	10030	9200	300	9710	10030	291	514660	149130	150
4.	8490	10000	295	9740	10100	305	530260	150850	110
5.	8010	10000	287	10050	9300	284	528420	44650	78
6.	10020	8500	263	10030	7350	279	400780	516930	145
7.	9630	10010	333	10000	8690	294	570980	115730	87
8.	9830	10030	327	8660	10030	285	518820	330770	169
9.	10070	7730	264	9940	10010	316	526100	35690	70
10.	10050	7000	264	10050	9090	311	509580	33570	75
Průměr	**9202,5**	**8918,45**	**283,9**	**8965,5**	**8841**	**273,65**	**511732**	**146716**	**104,7**
Počet výher	**5**	**5**		**4**	**6**		**9**	**1**	

Tabulka 6.4: Spolupracují agenti vs. Částečně spolupracující agenti.

- **Spolupracující agenti x Agenti bez spolupráce**

Další zkoumaný souboj skupiny agentů jsou **Splupracující agenti** (tým A) a **Agenti bez spolupráce** (tým B). Výsledky měření 10 běhů simulace jsou zobrazeny v tabulce **6.5**.

Jak je z tabulky vidět, v modelech A a B zvítězily téměř ve všech bězích agenti bez spolupráce. Důvod tohoto vítězství spočívá opět v rozmístění objektů na herní mřížce. Jelikož agenti bez spolupráce si nedokážou vyměňovat informace o svých cílech, většina agentů v týmu si na začátku simulace vybere stejný cíl. V případě modelu A je to již dříve zmiňovaný střed činnosti agentů, který je zvolen právě díky své pozici. Díky tomu že spolupracující agenti si volí různé cíle rovnoměrně po celé herní mřížce, agenti bez spolupráce díky značné přesile obsadí tuto klíčovou věž na středu. Vlastnictví této věže umožňuje agentům rychlý přístup ke kterémukoliv dalšímu prvku prostředí (věži, vlajce) a agenti tak získavají určitý náskok ve velikosti skóre, který je pro spolupracující agenty obtížné zdolat.

V případě modelu B dochází k jiným problémům. Protože zde není žádný střed zájmu agentů, nespolupracující agenti zde nemají výhodu jako v případě modelu B. V tomto případě je problémem samotná spolupráce, konkrétně zasílání požadavků, spolupracujících agentů. Tento problém je nejvíce patrný na obranných agentech, kteří často příjimají požadavky na obranu věže z různých částí herní mřížky od agentů vlajkami. Přestože přímá vzdálenost mezi těmito věžemi může být krátká, díky nutnosti obcházet zdi z překážek, je doba přesunu agenta značným způsobem prodloužena. Agent tak stráví většinu času přesunem mezi jednotlivými pozicemi, což je z pohledu výkonnosti týmu neefektivní a vede to k prohře týmu.

Situace v modelu C je v tomto případě podobná jako v předchozím případě v tabulce **6.4**. Spolupracující agenti zde dokáží využít vzájemných požadavků o podporu v boji a efektivně obsadit většinu věží rychleji než agenti bez spolupráce. Oproti předcházejícímu případu však stojí za povšimnutí průměrný počet tahů simulace, který je asi o 50 tahů větší. To je způsobeno tím, že nespolupracující agenti, díky tomu že většina si vybere stejnou věž, rychleji dokáží zabrat věže zpět.

		Model A			**Model B**			**Model C**	
	Tým A	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**	**Tým A**	**Tým B**	**Počet tahů**
1.	5110	10050	228	10030	8650	299	500580	285930	267
2.	6190	10040	250	7910	10030	287	526820	8470	76
3.	7870	10070	275	6650	10030	257	503140	181770	138
4.	4970	10030	232	4990	10090	226	507940	381770	227
5.	8900	10000	299	7320	10010	276	501540	140170	179
6.	5630	10010	255	6010	10290	259	509220	102570	148
7.	7550	10030	290	6340	10000	256	508980	142250	194
8.	5830	10020	236	7450	10030	274	538020	153850	145
9.	6990	10000	276	8760	10010	294	500420	166330	189
10.	5770	10010	255	7020	10010	276	581860	95830	135
Průměr	**6584**	**10023,5**	**262,8**	**6937,5**	**9975,5**	**262,75**	**517852**	**165894**	**169,8**
Počet výher	**0**	**10**		**1**	**9**		**10**	**0**	

Tabulka 6.5: Spolupracující agenti vs. Agenti bez spolupráce.

- **Částečně spolupracující agenti x Agenti bez spolupráce**

Poslední zkoumaný souboj jsou **Částečně spolupracující agenti** (tým A) a **Agenti bez spolupráce** (tým B). Výsledek testování je uveden v tabulce **6.6**.

Simulace modelu A má stejný výsledek jako v předchozím případě - tým agentů bez spolupráce získá převahnu nad soupeři díky rychlému obsazení střední pozice. Výsledky v modelu B se však výrazně liší. Částečně spolupracující agenti nemají implementovány metody pro zasílání požadavků a proto neztráci velké množství času přesunem po herní mřížce. To má za následek zvýšení efektity těchto agentů. Jak ukazuje hodnota průměrného skóre a počtu tahů, agenti jsou v efektivitě na přibližně stejné úrovni, tedy opět se do výsledku začíná projevovat náhodnost soubojů.

V modelu C je situace opět podobná předcházejícímu případu. Částečně spolupracující agenti dokáží díky informacím o cílech ostatních agentů zvolit cíle jiné a tím rychleji zabrat věže než agenti bez spolupráce. V tabulce je vidět, že, stejně jako v předchozím případě, je u těchto agentů poměr výher 10:0 a průměrné dosažené skóre mnohem větší.

		Model A				Model B				Model C	
	Tým A	**Tým B**	**Počet tahů**		**Tým A**	**Tým B**	**Počet tahů**		**Tým A**	**Tým B**	**Počet tahů**
1.	8480	10060	284		7700	10040	257		508580	167210	178
2.	5440	10010	230		7260	10030	278		501540	87370	142
3.	3820	10030	226		9790	10010	314		506660	113770	188
4.	8050	10000	282		7060	10020	268		508900	127690	177
5.	5820	10020	234		9530	10010	324		523780	220370	140
6.	5360	10060	235		10000	9820	332		508500	124170	191
7.	7430	10020	273		10000	9130	314		502500	137690	165
8.	9050	10030	307		10010	6230	276		503460	87690	166
9.	6040	10010	249		10010	5640	267		513060	152010	197
10.	6930	10020	254		6400	10070	270		524260	148130	175
Průměr	**6619,5**	**10027**	**258,9**		**8443,5**	**9205**	**282,8**		**510124**	**136610**	**171,9**
Počet výher	**0**	**10**			**4**	**6**			**10**	**0**	

Tabulka 6.6: Částečně spolupracující agenti vs. Agenti bez spolupráce

6.3 Shrnutí dosažených výsledků

Jak bylo ukázáno v kapitole **6.1** a **6.2** velmi velký vliv na efektivitu jednotlivých týmů agentů má konkétní umístění prvků v modelu prostředí. Jedná se zejména o:

- **Zvýšení celkové doby simulace**

Tuto situaci např. ilustruje tabulka **6.2** a **6.3**. Změna typu spolupráce z částečně spolupracujícího agenta na agenta bez spolupráce způsobí v jednom modelu prostředí (model C) velký nárůst doby provádění simulace, zatímco v jiném (model A a B) je nárůst minimální.

- **Změna celkové efektivity týmu**

Situace, kdy změna prostředí sníží (případně zvýší) efektivitu celého týmu. Např. v tabulce **6.6** je vidět, že v modelu A vyhrál každou simulaci tým agentů bez spolupráce, a v modelu C naopak všechny simulace prohrál.

Z hlediska implementace různých agentních systémů je tedy nutné přizpůsobit chování agentů v závislosti na řešeném problému. Jinak by se mohlo stát, že i kvalitně navržený agent bude podávat špatné výsledky.

7 Závěr

Cílem této práce bylo popsat návrh a implementenci simulátoru multiagentního systému. Simulátor je implementován jako abstraktní rozhraní pro připojení různých modelů prostředí. Program umožňuje do simulátoru vložit vlastní vytvořené agenty ve formě dynamické knihovny. Agenty je možné vkládat dynamicky za běhu simulace, bez nutnosti simulaci ukončit a znovu spustit. V rámci práce bylo vyhodnoceno chování několika typů agentů v různých modelech prostředí. Bylo ukázáno, že na výkonnost agenta má vliv mnoho různých faktorů, se kterými je nutné při návrhu multiagentního systému počítat.

Vzhledem k tomu, že simulátor byl implementován s důrazem na univerzálnost řešení, lze ho použít například pro výukové účely v kurzu Agentní a Multiagentní systémy (AGS). Studenti tohoto kurzu mohou využít simulátoru pro tvorbu a testování vlastních agentů na zadaném modelovém prostředí. Univerzálnost dále umožňuje rozšířit schopnosti simulátoru jako celku. Jako příklad je možné uvést implementaci síťové komunikace, kdy řízení simulace by mohlo probíhat na vzdáleném serveru. Klient by potom jednoduchým protokolem mohl zasílat požadavky a zpracoval by přijaté odpovědi. Další z možností jak rozšířit simulátor může být tvorba grafického uživatelského rozhraní pro ovládání simulace.

Literatura

[1] Bordini, R. H., Hübner, J. F.: *Jason: Java-based interpreter for an extended version of AgentSpeak* [online]. Release version 0.9.5, February 2007. Dostupné na URL: <http://jason.sourceforge.net/Jason.pdf>

[2] Bordini, R. H., Hübner, J. F.: *BDI agent programming in AgentSpeak using Jason* [online]. Dostupné na URL: <http://www.inf.ufrgs.br/~bordini/Publications/bapauj-LNCS-CR.pdf>

[3] Keil, D., Goldin, D.: *Indirect Interaction in Environments for Multiagent Systems* [online]. Springer 2006. Dostupné na URL: <http://www.engr.uconn.edu/~dqg/papers/e4mas.pdf>

[4] Rao, A., Geogeff, M. P.: *BDI agents: From Theory to Practice* [online]. Dostupné na URL: <https://www.aaai.org/Papers/ICMAS/1995/ICMAS95-042.pdf>

[5] Zbořil, F.: *Agentní a multiagentní systémy AGS (modul 1): Studijní opora* [online]. Verze 10/2006. Dostupné na URL: <http://perchta.fit.vutbr.cz:8000/vyuka-ags/uploads/3/AGS_opora_m1_v10r1.pdf>

[6] Zbořil, F.: *Agentní a multiagentní systémy AGS (modul 2): Studijní opora* [online]. Verze 10/2006. Dostupné na URL: <http://perchta.fit.vutbr.cz:8000/vyuka-ags/uploads/3/AGS_opora_m2_v10r1.pdf>

[7] Stødle, D.: *Creating pthreads in C++ using pointers to member functions* [online]. 3.9.2010. Dostupné na URL: <http://www.scsc.no/blog/2010/09-03-creating-pthreads-in-c++-using-pointers-to-member-functions.html>

[8] Haendel, L.: *The Function Pointer Tutorials: How to Implement Callbacks in C and C++?* [online]. 6.1.2005. Dostupné na URL: <http://www.newty.de/fpt/callback.html>

[9] Kilgard, J. M.: *The OpenGL Utility Toolkit (GLUT) Programming Interface* [online]. November 13, 1996. Dostupné na URL: <http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf>

[10] Turek, M.: *NeHe OpenGL Tutoriály* [online]. 29.2.2004. Dostupné na URL: <http://ii.iinfo.cz/r/k/cz_nehe_opengl.pdf>

Seznam příloh

Příloha 1. CD se zdrojovými soubory, dokumentací a plakátem